FWX
Perpetual Trading

Smart Contract Security Report

Final Report v1.0
August 20, 2024

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the `FWX Perpetual Trading` feature. This project was certified on `August 20, 2024`. The audit scope is limited to the `FWX Perpetual Trading` feature. Our security best practices strongly recommend that the `FWX team` conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About FWX Perpetual Trading feature

FWX Perpetual Trading is a decentralized perpetual exchange that allows users to trade various assets (such as cryptocurrencies, commodities, equities, and forex) with leverage directly from their deposited balance. It operates on EVM-based blockchain, providing a decentralized trading experience. FWX Perpetual Trading enables perpetual futures trading with up to 100x leverage. Traders can take both long and short positions based on dynamic pricing from PYTH Network. FWX Perpetual Trading allows users to use assets as collateral in a cross-margin system, providing flexibility in managing risk across different positions. In FWX Perpetual Trading's decentralized perpetual exchange, when a trader opens a leveraged position, they are essentially trading against the liquidity providers in the platform.

Liquidity Pool -- The FWX Perpetual Trading platform operates with a single-asset liquidity pool. Liquidity providers (LPs) deposit the asset into this pool, which is used to facilitate leveraged trading on the platform and LPs act as the counterparty to traders' positions.

Counterparty to the Trade -- The liquidity providers in the Liquidity Pool are the counterparties to the traders' positions. If a trader profits from a position, the payout comes from the liquidity pool, reducing its overall value. Conversely, if the trader incurs a loss, the liquidity pool benefits, as the loss is absorbed by the trader's collateral and added to the pool.

Fee Structure -- LPs earn fees from the trading activity on the platform, which includes trading and funding fees. These fees are distributed to LPs as a reward for providing liquidity and taking on the risk of being the counterparty to leveraged trades.

Project Summary

Item	Description
Client	FWX
Feature	FWX Perpetual Trading
Category	Decentralized Finance
Language	Solidity
Certified Date	August 20, 2024

Project Log

Description	Commit Hash
Initiate Audit	`56608b8767aa6ea021c829f26a008e3af84d3a8d (branch: feature/defi-perp)`
1st reassessment audit	`13ddbe5caf6b20da5b625ec7c4eaffd11fba91e8 (branch: feature/defi-perp)`
2st reassessment audit	`8575bdd6f269a647a17909df53545afcfa05f398 (branch: feature/defi-perp)`

Scope of Work

The security audit conducted does not replace the full security audit of the overall `FWX` protocol. The scope is limited to the `FWX Perpetual Trading` feature and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ PerpCore ▪ PerpLending ▪ PerpTrading ▪ PerpSetting ▪ PerpLib ▪ Imported associated smart contracts and libraries
Git Repository	▪ https://github.com/forward-x/defi-perp/
Audit Commit	▪ 56608b8767aa6ea021c829f26a008e3af84d3a8d (branch: feature/defi-perp)
Certified Commit	▪ 8575bdd6f269a647a17909df53545afcfa05f398 (branch: feature/defi-perp)
Audited Files	▪ contracts/src/perp/PerpCore.sol ▪ contracts/src/perp/PerpCoreBase.sol ▪ contracts/src/perp/PerpLending.sol ▪ contracts/src/perp/PerpLib.sol ▪ contracts/src/perp/PerpSetting.sol

	▪ contracts/src/perp/PerpTrading.sol ▪ contracts/interfaces/IPerpCore.sol ▪ contracts/interfaces/IPerpCoreBase.sol ▪ Other imported associated Solidity files
Excluded Files/Contracts	▪ contracts/src/perp/MockPerpCore.sol ▪ contracts/src/perp/PerpBot.sol ▪ contracts/src/perp/PerpHelper.sol

Remark: Our security best practices strongly recommend that the FWX team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	Anak Mirasing Kritsada Dechawattana Parichaya Thanawuthikrai Nattawat Songsom
Reviewers	Sumedt Jitpukdebodin

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software have no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, **Valix trusts that the** `FWX Perpetual Trading` feature has sufficient security protections to be safe for use.

Initially, `Valix identified 26 issues` during the assessment, categorized from "Critical" to "Informational" risk levels. The team's progress on these issues within the given timeframe is as follows:

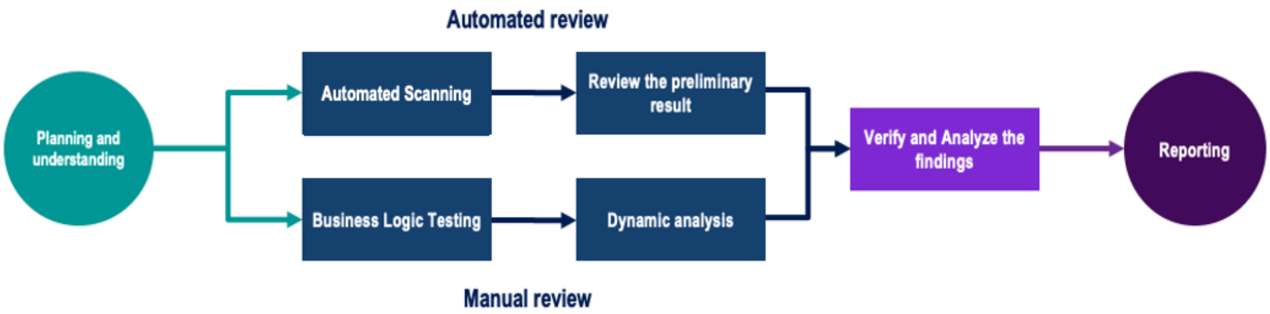
- 19 completely fixed issues
- 0 partially fixed issue
- 7 acknowledged issues

Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Risk Level	Findings Result	Status		
		● Fixed	● Partially Fixed	● Acknowledged
Critical	3	3	-	-
High	5	5	-	-
Medium	9	5	-	4
Low	4	2	-	2
Informational	5	4	-	1
Total	**26**	**19**	**0**	**7**

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application's purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
| Critical | C-xx	The code implementation does not match the specification, and it could disrupt the platform.
| High | H-xx	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
| Medium | M-xx	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
| Low | L-xx	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
| Informational | I-xx	Findings in this category are informational and may be further improved by following best practices and guidelines.

The risk value of each issue was calculated from the product of the **impact** and **likelihood** values, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Likelihood \ Impact	High	Medium	Low
High	• Critical	• High	• Medium
Medium	• High	• Medium	• Low
Low	• Medium	• Low	• Informational

The shading of the matrix visualizes the different risk levels.

Findings

System Trust Assumptions

Trust assumptions

The trust assumptions in this context are that the `FWX Perpetual Trading` protocol allows trusted managers to perform actions to manage critical functions and parameters with these following contracts:

- `PerpCore`
- `PerpSetting`
- `PerpTrading`

It is important to note that, while the trusted managers are granted specific privileges to oversee these contracts within the `FWX Perpetual Trading` protocol, special attention should be given to the accounts with these roles. These accounts have the authority to perform privileged actions such as setting crucial states and pausing/unpausing critical functions across the `FWX Perpetual Trading` protocol.

Furthermore, the trusted managers can execute actions with and without the need for a time-lock mechanism. This implies that any action within the scope of the trusted managers' authority will be carried out promptly, potentially affecting user positions and protocol parameters immediately.

Additional trust assumptions include:

- The liquidation process relies critically on an external `FWX` product to initiate timely liquidations, introducing a significant dependency on this system's reliability and effectiveness. Additionally, the protocol allows external entities to act as liquidators.
- The criteria for `liquidatePositionByTotalPnl` function are set off-chain, including factors such as PnL and contract size. This implies trust in the off-chain systems and processes that determine these criteria.
- There's trust placed in the managers to set spread configurations appropriately. These configurations are crucial to prevent arbitrageurs from profiting from price differences. The spread values must be carefully calculated based on acceptable price differences that could occur in the price feed from the Pyth oracle API. Improper configuration could lead to exploitation of the protocol.
- The protocol uses stablecoins (USDC) as collateral, while the prices throughout the protocol are compared using USD. There's an implicit assumption that the values of USDC and USD maintain a 1:1 ratio at all times. Any deviation from this peg could potentially impact the protocol's operations and user positions.

The privileged roles

In the `FWX Perpetual Trading` protocol, privileged roles have special access to perform sensitive actions, relying on the trust placed in these roles to ensure the proper functioning and security of the system.

The `PerpCore` contract:

- The `NoTimelockManager` account:
    - Can pause and unpause specific functions using `pause` and `unPause` functions, potentially halting critical operations like deposits, withdrawals, or trading.

The `PerpSetting` contract:

- The `ConfigTimelockManager` account:
    - Can set Pyth oracle IDs for tokens using `setPythId` function, determining the price feed sources for assets.
    - Can set the stale period for price feeds using `setStalePeriod` function, affecting when prices are considered outdated.
    - Can adjust bounty fee rates for the protocol and liquidators using `setBountyFeeRateToProtocol` and `setBountyFeeRateToLiquidator` functions.
    - Can set trading fees for specific underlying assets using `setTradingFee` function, directly affecting user costs.
    - Can set minimum and maximum open position sizes for assets using `setMinimumOpenSize` and `setMaximumOpenSize` functions.
    - Can modify margin ratios (maintenance and minimum) for assets using `setMarginRatio` function.
    - Can set the liquidity ratio using `setLiquidityRatio` function, which impacts the protocol's overall risk management.
    - Can set maximum PnL limits for assets using `setMaxPnls` function, potentially capping user profits.
    - Can set the liquidate PnL ratio using `setLiquidatePnlRatio` function, affecting when positions can be liquidated based on unrealized PnL.
    - Can configure open interest (OI) parameters including funding rates and spreads using `setOIConfig` function.

- The `AddressTimelockManager` account:
    - Can set the address of the `PerpTrading` contract using `setPerpTrading` function, potentially redirecting core trading functionality.

- Can set the address of the profit vault using `setProfitVaultAddress` function, determining where certain protocol fees are sent.
- Can set the fee-to-protocol rate using `setFeeToProtocolRate` function, affecting the distribution of fees between the protocol and liquidity providers.
- Can add or remove underlying assets from the allowed list using `setAllowUnderlying` function, controlling which assets can be traded on the platform.

The `PerpTrading` contract:

- The `NoTimelockManager` account:
  - Can liquidate positions based on total PnL using `liquidatePositionByTotalPnl` function, bypassing normal liquidation checks.
  - Can liquidate positions based on position PnL using `liquidatePositionByPositionPnl` function, bypassing normal liquidation checks.

These privileged roles in the `FWX Perpetual Trading` protocol have the power to significantly impact the system's operations, risk parameters, and user funds. The security and proper functioning of `FWX Perpetual Trading` rely on the trustworthiness and competence of the individuals or entities controlling these roles. Users of the `FWX Perpetual Trading` protocol are implicitly trusting these roles to manage the system responsibly and securely.

Review Findings Summary

The table below shows the summary of our assessments.

ID	Issue	Risk	Status
C-01	Incorrect Stale Price Flagging In _queryPythPrices Function	• Critical	Fixed
C-02	Incorrect Argument For The shortPNL Calculation	• Critical	Fixed
C-03	Incorrect Precision Handling In Liquidity Update When Withdrawing Liquidity	• Critical	Fixed
H-01	Excess Collateral Withdrawal Due To Decimal Discrepancies	• High	Fixed
H-02	Incorrect Validation Of Minimum Position Sizes In _validateOpenPositionInput Function	• High	Fixed
H-03	Inconsistency In Price Comparisons	• High	Fixed
H-04	Incorrect Open Interest Calculation In Funding Fee Determination	• High	Fixed
H-05	Inconsistency Between Calculated Global Unrealized PNL And Actual Value	• High	Fixed
M-01	Inconsistent Liquidation Check Across Function	• Medium	Fixed
M-02	Potential Use Of Stale Oracle Data In Deposit Function	• Medium	Fixed
M-03	Improper Handling Of Edge Cases In Pyth Price Calculation Function	• Medium	Fixed
M-04	Donation Attack To Increase atpPrice	• Medium	Acknowledged
M-05	Lack Of Price Protection Mechanisms In Position Opening	• Medium	Acknowledged
M-06	The totalMaintenanceMargin State Is Not Reset When All Positions Are Closed	• Medium	Fixed
M-07	Improperly Handle Liquidation Case When Closing Position	• Medium	Acknowledged
M-08	Improper Price Usage For Calculating Trading Fees	• Medium	Acknowledged
M-09	Over-Perturbed Entry Price Due To Initial Contract Size	• Medium	Fixed
L-01	Recommend Adhering To Best Practices For Confidence Interval Validation in Pyth Network Integration	• Low	Acknowledged
L-02	Using Weak Source Of Randomness For Calculate Price With	• Low	Acknowledged

	Spread		
L-03	Recommended Following Best Practices For Upgradeable Smart Contracts	• Low	Fixed
L-04	Incorrect Available Balance Validation For Open Position	• Low	Fixed
I-01	Typo: "maintainance" Used Instead Of "maintenance" Throughout The Codebase	• Informational	Fixed
I-02	Unnecessary Dummy Price Functions And State Variable	• Informational	Acknowledged
I-03	Recommended Removing Unused Code	• Informational	Fixed
I-04	Inconsistency Between Comment And Code	• Informational	Fixed
I-05	Mismatch Between Interface And Implementation For setMinimumOpenSize Function	• Informational	Fixed

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

Critical Risk

C-01 Incorrect Stale Price Flagging In _queryPythPrices Function

Risk	Likelihood	Impact	Status
● Critical	● High	● High	**Fixed**
Locations	PerpCoreBase.sol::_queryPythPrices L: 207 - 218		

Detailed Issue

The `_queryPythPrices` function in the `PerpCoreBase` contract contains a logical error that prevents the `isStale` flag from ever being set to `true`. This condition will only set `isStale` to `false` when `tempIsStale` is `false`. It will never set `isStale` to `true`.

This can lead to the use of stale prices in critical operations, potentially causing significant financial risks or vulnerabilities in the protocol.

```
File: PerpCoreBase.sol
207:  function _queryPythPrices(
208:      address[] memory tokens
209:  ) internal view returns (uint256[] memory prices, uint64[] memory publishTimes,
bool isStale) {
210:      prices = new uint256[](tokens.length);
211:      publishTimes = new uint64[](tokens.length);
212:      for (uint8 i = 0; i < tokens.length; i++) {
213:          bool tempIsStale;
214:          (prices[i], publishTimes[i], tempIsStale) = _queryPythPrice(tokens[i]);
215:
216:          if (!tempIsStale) isStale = tempIsStale;
217:      }
218:  }
```

Listing C-01.1 The `_queryPythPrices` function of the PerpCoreBase contract

Impact

This issue could allow the system to use outdated price data, which may result in:

- Incorrect valuation of positions
- Mispriced trades
- Improper liquidations
- Exploitation opportunities for malicious actors

Recommendations

We recommend removing the negation `(!)` from the condition. This change will correctly set the `isStale` flag when any price is stale.

```
File: PerpCoreBase.sol
function _queryPythPrices(
    address[] memory tokens
) internal view returns (uint256[] memory prices, uint64[] memory publishTimes, bool isStale) {
    prices = new uint256[](tokens.length);
    publishTimes = new uint64[](tokens.length);
    for (uint8 i = 0; i < tokens.length; i++) {
        bool tempIsStale;
        (prices[i], publishTimes[i], tempIsStale) = _queryPythPrice(tokens[i]);

        if (tempIsStale) isStale = tempIsStale;
    }
}
```

Listing C-01.2 The improved `_queryPythPrices` function of the PerpCoreBase contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

C-02 Incorrect Argument For The shortPNL Calculation

Risk	Likelihood	Impact	Status
● Critical	● High	● High	Fixed
Locations	PerpCoreBase.sol::_getAllUnRealizedPNL L: 114		

Detailed Issue

The _getAllUnRealizedPNL function in the PerpCoreBase contract calculates the total unrealized profit and loss. It does this by separately computing the values for both long (L107 - 111 in the code snippet C-02.1) and short (L112 - 116 in the code snippet C-02.1) positions, then summing them together.

However, for the short side, we noticed that the PerpLib._calculatePNL function is called with an incorrect argument. It passes temp.averagePriceLong (L114 in the code snippet C-02.1) instead of the temp.averagePriceShort to calculate the shortPNL.

```
File: PerpCoreBase.sol
100:     function _getAllUnRealizedPNL() internal view virtual returns (int256
result) {
101:         (uint256[] memory prices, , bool isStale) =
_queryPythPrices(allowUnderlyingList);
102:         require(!isStale, "PT/unrealize-pnl-stale");
103:
104:         for (uint8 i = 0; i < allowUnderlyingList.length; i++) {
105:             GlobalStat memory temp = globalStats[allowUnderlyingList[i]];
106:
107:             int256 longPNL = PerpLib._calculatePNL(
108:                 temp.totalContractSizeLong,
109:                 prices[i],
110:                 temp.averagePriceLong
111:             );
112:             int256 shortPNL = PerpLib._calculatePNL(
113:                 temp.totalContractSizeShort,
114:                 temp.averagePriceLong,
115:                 prices[i]
116:             );
117:             result += (longPNL + shortPNL);
118:         }
119:     }
```

Listing C-02.1 The _getAllUnRealizedPNL function of the PerpCoreBase contract

Impact

All subsequent calculations from the `_getAllUnRealizedPNL` function will be incorrect.

Recommendations

We recommend revising the `PerpLib._calculatePNL` function calling with the `temp.averagePriceShort` instead as shown in the code snippet below.

```
File: PerpCoreBase.sol
    function _getAllUnRealizedPNL() internal view virtual returns (int256
result) {
        (uint256[] memory prices, , bool isStale) =
_queryPythPrices(allowUnderlyingList);
        require(!isStale, "PT/unrealize-pnl-stale");

        for (uint8 i = 0; i < allowUnderlyingList.length; i++) {
            GlobalStat memory temp = globalStats[allowUnderlyingList[i]];

            int256 longPNL = PerpLib._calculatePNL(
                temp.totalContractSizeLong,
                prices[i],
                temp.averagePriceLong
            );
            int256 shortPNL = PerpLib._calculatePNL(
                temp.totalContractSizeShort,
                temp.averagePriceShort,
                prices[i]
            );
            result += (longPNL + shortPNL);
        }
    }
```

Listing C-02.2 The improved `_getAllUnRealizedPNL` function of the PerpCoreBase contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The FWX team adopted our recommendation and fixed this issue.

C-03 Incorrect Precision Handling In Liquidity Update When Withdrawing Liquidity

Risk	Likelihood	Impact	Status
● Critical	● High	● High	**Fixed**
Locations	PerpLending.sol::_withdraw L: 72		

Detailed Issue

The calculation of input parameters passed into the `_updateLiquidity` function is incorrect when the collateral token does not have 18 decimals. The result will be returned in the `COLLATERAL` precision:

```
(atpBurnAmount * atpPrice) / WEI_UNIT;
COLLATERAL_UNIT + WEI_UNIT - WEI_UNIT = COLLATERAL_UNIT
```

However, since `liquidity` is maintained in `WEI_UNIT`, this causes a precision mismatch in the calculation. As a result, the `_updateLiquidity` function returns an incorrect input amount, which affects the subsequent `withdrawAmount` unit conversion (L75), causing it to be rounded down and preventing any liquidity withdrawal while the ATP of the holder is burned.

```
File: PerpLending.sol
51:     function _withdraw(uint256 nftId, uint256 withdrawAmount) internal {
52:         PoolTokens storage tokenHolder = tokenHolders[nftId];
---
72:         withdrawAmount = _updateLiquidity((atpBurnAmount * atpPrice) / WEI_UNIT,
false);
---
74:         // update withdrawAmount to 1eColla
75:         withdrawAmount = (withdrawAmount * COLLATERAL_UNIT) / WEI_UNIT;
76:         _transferOut(msg.sender, tokenAddress, withdrawAmount);
77:
78:         emit Withdraw(msg.sender, nftId, withdrawAmount, pBurnAmount,
atpBurnAmount, 0, 0, 0);
79:     }
```

Listing C-03.1 The `_withdraw` function of the `PerpLending` contract

```
File: PerpCoreBase.sol
142:     function _updateLiquidity(uint256 amount, bool isAdd) internal returns
(uint256) {
143:         if (isAdd) {
144:             liquidity += amount;
145:         } else {
146:             amount = MathUpgradeable.min(liquidity, amount);
```

```
                liquidity -= amount;
            }

            return amount;
        }
```

Listing C-03.2 The `_updateLiquidity` function of the `PerpCoreBase` contract

Impact

Incorrect precision in the liquidity update calculation affects the subsequent `withdrawAmount` unit conversion, leading to it being rounded down and resulting in the inability to withdraw any liquidity while the holder's ATP is burned.

Scenario

1. The lender deposits the `100 USDC (100e6)` as `liquidity`
    a. `tokenHolder.pToken` = 100e6
    b. `tokenHolder.atpToken` = 100e6
    c. `liquidity` = 0 + 100e6 * WEI_UNI(1e18) / COLLATERAL_UNIT(1e6) = 100e18
2. The lender withdraws all their liquidity: `100 USDC (100e6)` (Assuming `atpPrice` remains stable at `WEI_UNIT`):
    a. `tokenHolder.pToken` = 100e6 - 100e6 (`pBurnAmount`) = 0
    b. `tokenHolder.atpToken` = 100e6 - 100e6 (`atpBurnAmount`) = 0
    c. Input amount in the `_updateLiquidity` function:

        `100e6 * WEI_UNI / WEI_UNI = 100e6`

    d. Liquidity incorrectly updated:

        `liquidity: 100e18 - amount: 100e6`

    e. Incorrect value returned as `withdrawAmount` (L72): 100e6
    f. Attempted to convert `withdrawAmount` to 18-decimal precision (L75):

        = `100e6 * COLLATERAL_UNIT(1e6) / WEI_UNI(1e18)`

        = 100e12 / 1e18 = 1e14 / 1e18 = **0**

    g. Transfer of 0 withdraws amount:

        `_transferOut(msg.sender, tokenAddress, withdrawAmount: 0);`

Recommendations

We recommend updating the precision conversion of the input amount with the following code snippet. This ensures that the input amount precision will be used in further calculations as `WEI_UNIT`:

```
(atpBurnAmount * atpPrice) / COLLATERAL_UNIT;
COLLATERAL_UNIT + WEI_UNIT - COLLATERAL_UNIT = WEI_UNIT
```

```
File: PerpLending.sol
    function _withdraw(uint256 nftId, uint256 withdrawAmount) internal {
        PoolTokens storage tokenHolder = tokenHolders[nftId];
---
        withdrawAmount = _updateLiquidity((atpBurnAmount * atpPrice) /
COLLATERAL_UNIT, false);
---
        // update withdrawAmount to 1eColla
        withdrawAmount = (withdrawAmount * COLLATERAL_UNIT) / WEI_UNIT;
        _transferOut(msg.sender, tokenAddress, withdrawAmount);

        emit Withdraw(msg.sender, nftId, withdrawAmount, pBurnAmount,
atpBurnAmount, 0, 0, 0);
    }
```

Listing C-03.3 The improved `_withdraw` function of the `PerpLending` contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

High Risk

H-01 Excess Collateral Withdrawal Due To Decimal Discrepancies

Risk	Likelihood	Impact	Status
● High	● Medium	● High	Fixed
Locations	PerpTrading.sol::_withdrawCollateral L: 145		

Detailed Issue

We identified a mismatch in precision when comparing the minimum withdrawal collateral amount with the available balance retrieved from the `_getBalance` function (L143).

To elaborate, the `_getBalance` function calculates and returns the `netBalance` and `availableBalance` with 18 decimal precision. However, if the collateral token does not use 18 decimal precision, this discrepancy can lead to incorrect comparisons with the `availableBalance` value in the `_withdrawCollateral` function.

```
File: PerpTrading.sol
136:     function _withdrawCollateral(
137:         uint256 nftId,
138:         address collateralToken,
139:         address underlyingToken,
140:         uint256 amount
141:     ) internal {
142:         uint256 wallet = wallets[nftId][0];
143:         (, uint256 availableWallet) = _getBalance(nftId);
144:         amount = MathUpgradeable.min(amount, wallet);
145:         amount = MathUpgradeable.min(amount, availableWallet);
146:
147:         wallets[nftId][0] -= amount;
148:
149:         _transferOut(msg.sender, collateralToken, amount);
150:
151:         emit UpdateWallet(msg.sender, nftId, 0, wallets[nftId][0] + amount,
wallets[nftId][0]);
152:         emit WithdrawCollateral(msg.sender, nftId, collateralToken,
underlyingToken, 0, amount);
153:     }
```

Listing H-01.1 The `_withdrawCollateral` function of the `PerpTrading` contract

```
File: PerpTrading.sol
    function _getBalance(
        uint256 nftId
    ) internal view returns (int256 netBalance, uint256 availableBalance) {
        uint256 wallet = PerpLib._toWeiUnit(wallets[nftId][0], COLLATERAL_UNIT);
        (int256 unrealizedPnl, uint256 tradingFee, uint256 fundingFee) =
_getUnrealizedPnlAndFee(
            nftId
        );
        NftStat memory nftStat = nftStats[nftId];
        // NOTE: net balance = wallet + 'äëcollaLocled + 'äëpnl - 'äëfee
        netBalance =
            int256(wallet + nftStat.totalCollateralLocked) -
            int256(tradingFee + fundingFee) +
            (unrealizedPnl + nftStat.unsettlePnl);

        // NOTE: available balance = wallet + 'äëpnl - 'äëfee

        availableBalance = PerpLib._toUint(netBalance -
int256(nftStat.totalCollateralLocked));
        availableBalance = MathUpgradeable.min(availableBalance, wallet);
    }
```

Listing H-01.2 The `_getBalance` function of the `PerpTrading` contract

Impact

Incorrectly updates the wallet, allowing traders to withdraw more than the available balance, which should remain unwithdrawable to secure the position.

Scenario

Support that there is no fee calculation included in the scenario.

1. The trader deposits the `1000 USDC (1000e6)` as `collateral`
    a. Trader's wallet = 1000e6
2. The trader opens a long position and provides `600 USDC (600e6)` as `collateral`
    a. Trader's wallet = 1000e6 - 600e6 = 400e6
    b. Trader's locked collateral: 600e6
3. The trader immediately withdraws their remaining wallet balance: 400e6
    a. `availableWallet` = `wallet + ∑pnl - ∑fee`

        = 400e18 + (-1e18)(some pnl) - 0

        = 400e18 - 10e18

        = **390e18**

4. The withdrawal amount is bound against the available wallet (L145):

   `amount = MathUpgradeable.min(amount: 400e6, availableWallet: 390e18);`

   `amount` = **400e6**

As shown in the scenario above, the expected available wallet that should be allowed for withdrawal is 390 USDC, but the trader can withdraw 400 USDC.

Recommendations

We recommend updating the `_withdrawCollateral` function to convert the `availableWallet` value to the appropriate precision before comparing it with the specified withdrawal amount.

```
File: PerpTrading.sol
136:     function _withdrawCollateral(
137:         uint256 nftId,
138:         address collateralToken,
139:         address underlyingToken,
140:         uint256 amount
141:     ) internal {
142:         uint256 wallet = wallets[nftId][0];
143:         (, uint256 availableWallet) = _getBalance(nftId);
144:         amount = MathUpgradeable.min(amount, wallet);
145:         amount = MathUpgradeable.min(amount,
PerpLib._toCollateralUnit(availableWallet, COLLATERAL_UNIT));
146:
147:         wallets[nftId][0] -= amount;
148:
149:         _transferOut(msg.sender, collateralToken, amount);
150:
151:         emit UpdateWallet(msg.sender, nftId, 0, wallets[nftId][0] + amount,
wallets[nftId][0]);
152:         emit WithdrawCollateral(msg.sender, nftId, collateralToken,
underlyingToken, 0, amount);
153:     }
```

Listing H-01.3 The improved `_withdrawCollateral` function of the `PerpTrading` contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

H-02 Incorrect Validation Of Minimum Position Sizes In _validateOpenPositionInput Function

Risk	Likelihood	Impact	Status
● High	● High	● Medium	Fixed
Locations	PerpTrading.sol::_validateOpenPositionInput L: 697 - 712 PerpTrading.sol::_openPosition L: 207		

Detailed Issue

We found that the `_validateOpenPositionInput` function in the `PerpTrading` contract is not correctly validating the minimum open size for position. The current implementation uses the total notional value (existing position + new position) for both minimum and maximum size checks.

However, according to the requirements from `FWX` team:

- **The minimum open size should be validated for each individual position opening**, ensuring it's greater than or equal to the `minimumOpenSize` state.
- **The maximum open size should be validated using the total position size (new opening position + existing position)**, ensuring it's less than or equal to the `maximumOpenSize` state.

This incorrect validation allows users to open positions smaller than the minimum size requirement.

```
File: PerpTrading.sol
204:     _validateOpenPositionInput(
205:         collateralToken,
206:         underlyingToken,
207:         ((pos.contractSize * pos.entryPrice) + (contractSize * temp.entryPrice))
/ WEI_UNIT,
208:         leverage
209:     );
```

Listing H-02.1 The `_openPosition` function of the PerpTrading contract

```
File: PerpTrading.sol
697:     function _validateOpenPositionInput(
698:         address collateralToken,
699:         address underlyingToken,
700:         uint256 notional,
701:         uint256 leverage
702:     ) internal view {
703:         require(collateralToken == tokenAddress, "PT/collateral-invalid");
704:         require(allowUnderlying[underlyingToken], "PT/token-not-allowed");
705:         require(notional <= maximumOpenSize[underlyingToken],
"PT/contract-size-more-than-maximum");
706:         require(notional >= minimumOpenSize[underlyingToken],
```

```
"PT/contract-size-less-than-minimum");
            require(
                leverage >= WEI_UNIT &&
                    (WEI_PERCENT_UNIT * WEI_UNIT) / leverage >=
minimumMarginRatio[underlyingToken],
                "PT/invalid-leverage"
            );
        }
```

Listing H-02.2 The `_validateOpenPositionInput` function of the PerpTrading contract

Impact

The current implementation allows positions to be opened that might not meet the intended minimum size requirements when considered individually.

This could significantly affect the risk management and operational parameters of the protocol, **leading to positions too small.** This may increase systemic risk, potentially impact the economic model of the protocol, and compromise the overall stability of the system.

Recommendations

We recommended revising the `_openPosition` and `_validateOpenPositionInput` functions in the `PerpTrading` contract to correctly implement the size validations. The changes should be as follows:

```
File: PerpTrading.sol
        function _openPosition(
            uint256 nftId,
            bool isLong,
            address collateralToken,
            address underlyingToken,
            uint256 contractSize,
            uint256 leverage
        ) internal {
---
            _validateOpenPositionInput(
                collateralToken,
                underlyingToken,
                (contractSize * temp.entryPrice) / WEI_UNIT,
                (pos.contractSize * pos.entryPrice) / WEI_UNIT,
                leverage
            );
---
        }
```

Listing H-02.3 The improved `_openPosition` function of the PerpTrading contract

Then, update the `_validateOpenPositionInput` function:

```
File: PerpTrading.sol
    function _validateOpenPositionInput(
        address collateralToken,
        address underlyingToken,
        uint256 newPositionNotional,
        uint256 existingPositionNotional,
        uint256 leverage
    ) internal view {
        require(collateralToken == tokenAddress, "PT/collateral-invalid");
        require(allowUnderlying[underlyingToken], "PT/token-not-allowed");
        require(newPositionNotional >= minimumOpenSize[underlyingToken], "PT/contract-size-less-than-minimum");
        require(newPositionNotional + existingPositionNotional <= maximumOpenSize[underlyingToken], "PT/contract-size-more-than-maximum");
        require(
            leverage >= WEI_UNIT &&
                (WEI_PERCENT_UNIT * WEI_UNIT) / leverage >= minimumMarginRatio[underlyingToken],
            "PT/invalid-leverage"
        );
    }
```

Listing H-02.4 The improved `_validateOpenPositionInput` function of the PerpTrading contract

These changes ensure that the minimum size check is performed only on the new position, while the maximum size check considers the total position size, including any existing position.

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

H-03 Inconsistency In Price Comparisons

Risk	Likelihood	Impact	Status
● High	● High	● Medium	**Fixed**
Locations	PerpTrading.sol::_openPosition L: 155 - 278 PerpTrading.sol::_isLiquidableByTotalPnl L: 487 - 491 PerpLending.sol::_getInterestTokenPrice L: 84 - 94		

Detailed Issue

The protocol provided two functions to fetch the prices in different unit

- The `_queryPythPrice` function (code snippet H-03.1) fetches the given price in USD
- The `_getSafeCurrentPrice` function (code snippet H-03.2) fetches the given price in the collateral unit

We discovered that there are inconsistencies in price comparisons and calculations from the result of the mentioned functions:

1. Within the `_openPosition` function (code snippet H-03.3), we discovered inconsistencies in price unit usage. Some calculations (L166, 216, 217, 223, and 227 in code snippet H-03.3) **use the price in the collateral unit**, while validations (L234, and 238 in code snippet H-03.3) **uses both the collateral unit and USD price for comparison**.
2. Within the `_isLiquidableByTotalPnl` function (code snippet H-03.6), we found a similar issue where the validation uses both the collateral unit and USD price for comparison (L490 in code snippet H-03.6)
3. Within the `_getInterestTokenPrice` function (code snippet H-03.7). This function calculates the interest price using liquidity in the collateral unit (L89 in code snippet H-03.7) and traderPnl in USD (L85 and 91 in code snippet H-03.7), leading to incorrect calculations during the deposit and withdrawal processes of lending.

```
File: PerpCoreBase.sol
196:     function _queryPythPrice(
197:         address token
198:     ) internal view virtual returns (uint256 price, uint64 publishTime, bool
isStale) {
199:         PythStructs.Price memory pythPrice =
pyth.getPriceUnsafe(pythOracleId[token]);
200:         price = PerpLib._calculatePythPrice(pythPrice);
201:         publishTime = uint64(pythPrice.publishTime);
202:         if (block.timestamp > stalePeriod + publishTime) {
203:             isStale = true;
```

```
        }
    }

    function _queryPythPrices(
        address[] memory tokens
    ) internal view returns (uint256[] memory prices, uint64[] memory publishTimes, bool isStale) {
        prices = new uint256[](tokens.length);
        publishTimes = new uint64[](tokens.length);
        for (uint8 i = 0; i < tokens.length; i++) {
            bool tempIsStale;
            (prices[i], publishTimes[i], tempIsStale) = _queryPythPrice(tokens[i]);

            if (!tempIsStale) isStale = tempIsStale;
        }
    }
```

Listing H-03.1 The `_queryPythPrice` function of the `PerpCoreBase` contract

```
File: PerpTrading.sol
    function _getSafeCurrentPrice(
        address underlyingToken
    ) internal view virtual returns (uint256 currentPrice, uint64 publishTime) {
        bool isStale;
        uint256 underPrice;
        uint256 underPublishTime;
        (underPrice, underPublishTime, isStale) = _queryPythPrice(underlyingToken);
        require(!isStale, "PT/close-position-pnl-stale");

        uint256 collaPrice = WEI_UNIT;
        uint256 collaPublishTime;
        (collaPrice, collaPublishTime, isStale) = _queryPythPrice(tokenAddress);
        require(!isStale, "PT/close-position-pnl-stale");
        return (
            (underPrice * WEI_UNIT) / collaPrice,
            uint64(MathUpgradeable.min(underPublishTime, collaPublishTime))
        );
    }
```

Listing H-03.2 The `_getSafeCurrentPrice` function of the PerpTrading contract

```
File: PerpTrading.sol
    function _openPosition(
        uint256 nftId,
        bool isLong,
        address collateralToken,
        address underlyingToken,
        uint256 contractSize,
        uint256 leverage
    ) internal {
        PerpLib.OpenPositionTempStruct memory temp;
```

```
        {
            uint64 publishTime;
            (temp.entryPrice, publishTime) = _getSafeCurrentPrice(underlyingToken); // USDC
            temp.entryPrice = PerpLib._calculatePriceWithSpread( // USDC
                temp.entryPrice,
                contractSize,
                isLong,
                publishTime,
                stalePeriod,
                spreadNotional[underlyingToken],
                spread[underlyingToken]
            );
        }
---
        temp.notional = (temp.contractSize * temp.entryPrice) / WEI_UNIT; // USDC
        temp.collateral = (temp.notional * WEI_UNIT) / leverage; // USDC
        temp.maintenanceMargin = PerpLib._calPercentage(
            temp.notional,
            maintainanceMarginRatio[temp.underlyingToken]
        );

        temp.tradingFee = PerpLib._calPercentage( // USDC
            temp.notional,
            tradingFeeRates[temp.underlyingToken]
        );
        temp.fundingFee = _getFundingFee( // USDC
            temp.isLong,
            temp.contractSize,
            temp.entryPrice,
            temp.underlyingToken
        );

        _validateLiquidityRatio(temp.isLong, temp.notional); // validate in different unit

        (, uint256 availableBalance) = _getBalance(nftId);
        require(
            availableBalance > temp.collateral + temp.tradingFee + temp.fundingFee,
            "PT/insufficient-wallet-balance"
        );
```

Listing H-03.3 The `_openPosition` function of the PerpTrading contract

```
File: PerpTrading.sol
    function _getTotalOI() internal view returns (uint256 totalOILong, uint256 totalOIShort) {
        (uint256[] memory prices, , bool isStale) = _queryPythPrices(allowUnderlyingList); // USD
        require(!isStale, "PT/unrealize-pnl-stale");

```

```
561:         for (uint8 i = 0; i < allowUnderlyingList.length; i++) {
562:             GlobalStat memory stat = globalStats[allowUnderlyingList[i]];
563:
564:             totalOILong += (stat.totalContractSizeLong * prices[i]) / WEI_UNIT;
565:             totalOIShort += (stat.totalContractSizeShort * prices[i]) / WEI_UNIT;
566:         }
567:     }
---
714:     function _validateLiquidityRatio(bool isLong, uint256 newNotional) internal view {
715:         (uint256 totalOILong, uint256 totalOIShort) = _getTotalOI(); // USD
716:         if (isLong) totalOILong += newNotional;
717:         else totalOIShort += newNotional;
718:
719:         uint256 netOI = totalOILong > totalOIShort
720:             ? (totalOILong - totalOIShort)
721:             : (totalOIShort - totalOILong);
722:         uint256 currentLiquidityRatio = (netOI * WEI_UNIT) / liquidity;
723:         require(currentLiquidityRatio <= liquidityRatio, "PT/not-enough-liquidity");
724:     }
```

Listing H-03.4 The `_validateLiquidityRatio` function of the PerpTrading contract

```
File: PerpTrading.sol
760:     function _getBalance(
761:         uint256 nftId
762:     ) internal view returns (int256 netBalance, uint256 availableBalance) {
763:         uint256 wallet = PerpLib._toWeiUnit(wallets[nftId][0], COLLATERAL_UNIT); // USDC
764:         (int256 unrealizedPnl, uint256 tradingFee, uint256 fundingFee) = _getUnrealizedPnlAndFee(
765:             nftId
766:         ); // USD
767:         NftStat memory nftStat = nftStats[nftId];
768:         // NOTE: net balance = wallet + ∑collaLocled + ∑pnl - ∑fee
769:         netBalance =
770:             int256(wallet + nftStat.totalCollateralLocked) -
771:             int256(tradingFee + fundingFee) +
772:             (unrealizedPnl + nftStat.unsettlePnl);
773:
774:         // NOTE: available balance = wallet + ∑pnl - ∑fee
775:
776:         availableBalance = PerpLib._toUint(netBalance - int256(nftStat.totalCollateralLocked));
777:         availableBalance = MathUpgradeable.min(availableBalance, wallet);
778:     }
779:
780:     function _getUnrealizedPnlAndFee(
781:         uint256 nftId
782:     ) internal view returns (int256 unrealizedPnl, uint256 tradingFee, uint256 fundingFee) {
```

```
        (uint256[] memory prices, uint64[] memory publishTimes, bool isStale) = _queryPythPrices(
            allowUnderlyingList
        ); // USD
        require(!isStale, "PT/unrealize-pnl-stale");

        for (uint8 i = 0; i < allowUnderlyingList.length; i++) {
            bytes32 pairByte = PerpLib._hashPair(tokenAddress, allowUnderlyingList[i]);
            Position memory pos = positions[nftId][pairByte];

            if (pos.id == 0) continue;

            PositionState memory posState = positionStates[nftId][pos.id];
            uint256 perturbPrice = PerpLib._calculatePriceWithSpread(
                prices[i],
                pos.contractSize,
                !posState.isLong,
                publishTimes[i],
                stalePeriod,
                spreadNotional[pos.underlyingToken],
                spread[pos.underlyingToken]
            );
            int256 pnl = posState.isLong
                ? PerpLib._calculatePNL(pos.contractSize, perturbPrice, pos.entryPrice)
                : PerpLib._calculatePNL(pos.contractSize, pos.entryPrice, perturbPrice);

            unrealizedPnl += pnl > maxPnls[pos.underlyingToken]
                ? maxPnls[pos.underlyingToken]
                : pnl;
            tradingFee +=
                (perturbPrice * pos.contractSize * tradingFeeRates[pos.underlyingToken]) /
                (WEI_UNIT * WEI_PERCENT_UNIT);
            fundingFee += _getFundingFee(
                !posState.isLong,
                pos.contractSize,
                perturbPrice,
                pos.underlyingToken
            );
        }
    }
```

Listing H-03.5 The _getBalance function of the PerpTrading contract

```
File: PerpTrading.sol
    function _isLiquidableByTotalPnl() internal view returns (bool isPosLiquidable) {
        int256 totalPnl = _getAllUnRealizedPNL();
        isPosLiquidable =
            (liquidity * liquidatePnlRatio) / WEI_PERCENT_UNIT <=
```

```
PerpLib._toUint(totalPnl);
    }
```

Listing H-03.6 The `_isLiquidableByTotalPnl` function of the PerpTrading contract

```
File: PerpLending.sol
    function _getInterestTokenPrice() internal view returns (uint256 price) {
        int256 traderPnl = _getAllUnRealizedPNL();

        price = uint256(
            PerpLib._calculateInterestTokenPrice(
                int256(liquidity),
                int256((atpTokenTotalSupply * WEI_UNIT) / COLLATERAL_UNIT),
                traderPnl,
                int256(WEI_UNIT)
            )
        );
    }
```

Listing H-03.7 The `_getInterestTokenPrice` function of the PerpLending contract

```
File: PerpCoreBase.sol
    function _getAllUnRealizedPNL() internal view virtual returns (int256 result) {
        (uint256[] memory prices, , bool isStale) = _queryPythPrices(allowUnderlyingList);
        require(!isStale, "PT/unrealize-pnl-stale");

        for (uint8 i = 0; i < allowUnderlyingList.length; i++) {
            GlobalStat memory temp = globalStats[allowUnderlyingList[i]];

            int256 longPNL = PerpLib._calculatePNL(
                temp.totalContractSizeLong,
                prices[i],
                temp.averagePriceLong
            );
            int256 shortPNL = PerpLib._calculatePNL(
                temp.totalContractSizeShort,
                temp.averagePriceLong,
                prices[i]
            );
            result += (longPNL + shortPNL);
        }
    }
```

Listing H-03.8 The `_getAllUnRealizedPNL` function of the PerpCoreBase contract

Impact

Impact for open position:

This inconsistency could lead to incorrect available balance validating. Moreover, the impact of this issue depends on the collateral price in USD.

If the current collateral price is under the USD price, the returned `availableBalance` from `_getBalance` function is greater than their actual available balance.

Conversely, if the current collateral price is over the USD price, the returned `availableBalance` from `_getBalance` function is lower than their actual available balance.

Impact for liquidation:

If the current collateral price is under the USD price, the position will be further from the liquidation criteria than it should be. This scenario disadvantages lenders and protocol.

Conversely, if the current collateral price is over the USD price, the position will be closer to the liquidation criteria than it should be. In this case, traders are at a disadvantage.

Impact for the `_getInterestTokenPrice` function:

The inconsistency leading to incorrect calculations during the deposit and withdrawal processes of lending.

Recommendations

For the recommendation, we advise the `FWX` team to maintain consistency in units across states, settings, and calculations. This approach will ensure that the implementation aligns with the business requirements.

Reassessment

The `FWX` team addressed this issue by revising the problematic functions, by converting prices to collateral units before using them in comparisons and calculations. This approach will align with the business requirements.

H-04 Incorrect Open Interest Calculation In Funding Fee Determination

Risk	Likelihood	Impact	Status
• High	• High	• Medium	**Fixed**
Locations	PerpTrading.sol::_openPosition L: 230 PerpCoreBase.sol::_getFundingFee L: 121 - 140 PerpLib.sol::_getFundingFeeRate L: 116 - 140		

Detailed Issue

We found that the `_getFundingFee` function in the `PerpCoreBase` contract is calculating the Open Interest (OI) incorrectly when a user opens a position. This function is called as part of the position opening process to determine the funding fee. The current implementation passes the OI as a measure of contract sizes to the `PerpLib._getFundingRate` function, without accounting for the asset's price (L135 in code snippet H-04.2).

However, the `PerpLib.getFundingRate` function expects OI in terms of value, not just contract size.

This discrepancy leads to inaccurate funding rate calculations, as the true value of open positions is not being considered. Consequently, users opening positions are charged incorrect funding fees, which can be either too high or too low depending on the current price of the asset.

```
File: PerpTrading.sol
155:     function _openPosition(
156:         uint256 nftId,
157:         bool isLong,
158:         address collateralToken,
159:         address underlyingToken,
160:         uint256 contractSize,
161:         uint256 leverage
162:     ) internal {
---
227:         temp.fundingFee = _getFundingFee(
228:             temp.isLong,
229:             temp.contractSize,
230:             temp.entryPrice,
231:             temp.underlyingToken
232:         );
---
278:     }
```

Listing H-04.1 The `_openPosition` function of the PerpTrading contract

```
File: PerpCoreBase.sol
    function _getFundingFee(
        bool isBuy,
        uint256 contractSize,
        uint256 price,
        address underlyingToken
    ) internal view returns (uint256 fundingFee) {
        GlobalStat memory stat = globalStats[underlyingToken];
        int256 currentOI = PerpLib._calculateOI(
            int256(stat.totalContractSizeLong + (isBuy ? contractSize : 0)),
            int256(stat.totalContractSizeShort + (!isBuy ? contractSize : 0))
        );
        uint256 fundingRate = PerpLib._getFundingRate(
            fundingNetOI[underlyingToken],
            fundingRates[underlyingToken],
            currentOI,
            isBuy
        );

        fundingFee = PerpLib._calPercentage((contractSize * price) / WEI_UNIT, fundingRate);
    }
```

Listing H-04.2 The `_getFundingFee` function of the PerpCoreBase contract

```
File: PerpLib.sol
    function _getFundingRate(
        uint256[] memory fundingNetOI, // open interest of short or long
        uint256[] memory fundingRates,
        int256 oi, // + long > short, - short < long in % unit  (currentOIRatio)
        bool isBuy // isBuy -> open long, close short, !isBuy -> open short, close long
    ) internal pure returns (uint256 fundingRate) {
        // if new open not in the same current oi ratio, no funding rate
        if ((isBuy && oi <= 0) || (!isBuy && oi >= 0)) {
            return fundingRate;
        }

        oi = _abs(oi);
        for (uint8 i = 1; i < fundingNetOI.length; i++) {
            if (uint256(oi) > fundingNetOI[i]) continue;

            uint256 r1 = fundingRates[i - 1];
            uint256 r2 = fundingRates[i];
            uint256 u1 = fundingNetOI[i - 1];
            uint256 u2 = fundingNetOI[i];
            uint256 diffOI = uint256(oi) - u1;
            return r1 + (diffOI * (r2 - r1)) / (u2 - u1);
        }

        return fundingRates[fundingNetOI.length - 1];
    }
```

Listing H-04.3 The `_getFundingRate` function of the PerpLib library

Impact

Incorrect funding fee calculations occur when users open positions, leading to unfair fee charges. This can result in financial losses for users and disrupt the protocol's economic balance. The issue could be exploited by traders, especially with large positions or during high price volatility, potentially destabilizing the protocol's funding mechanism.

Recommendations

We recommend modifying the `_getFundingFee` function to calculate the OI value by multiplying the current OI with the asset price before passing it to the `PerpLib._getFundingRate` function. The following change should be implemented:

```
File: PerpCoreBase.sol
    function _getFundingFee(
---
        int256 currentOIValue = (currentOI * int256(price)) / int256(WEI_UNIT);
        uint256 fundingRate = PerpLib._getFundingRate(
            fundingNetOI[underlyingToken],
            fundingRates[underlyingToken],
            currentOIValue,
            isBuy
        );
---
    }
```

Listing H-04.4 The `_getFundingFee` function of the PerpCoreBase contract

This adjustment ensures that the funding rate is calculated based on the actual value of open interest, providing a more accurate and fair funding fee mechanism when users open positions. After implementing this change, thorough testing should be conducted to verify the correct behavior of funding rate calculations across various market conditions and position sizes.

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The FWX team adopted our recommendation and fixed this issue by revising the _getFundingRate and _getFundingFee functions.

```
File: PerpCoreBase.sol
function _getFundingRate(
    bool isBuy,
    uint256 contractSize,
    uint256 price,
    address underlyingToken
) internal view returns (uint256 fundingRate) {
    GlobalStat memory stat = globalStats[underlyingToken];
    int256 currentOI = PerpLib._calculateOI(
        int256(stat.totalContractSizeLong + (isBuy ? contractSize : 0)),
        int256(stat.totalContractSizeShort + (!isBuy ? contractSize : 0))
    );
    fundingRate = PerpLib._getFundingRate(
        fundingNetOI[underlyingToken],
        fundingRates[underlyingToken],
        (currentOI * int256(price)) / int256(WEI_UNIT),
        isBuy
    );
}

function _getFundingFee(
    bool isBuy,
    uint256 contractSize,
    uint256 price,
    address underlyingToken
) internal view returns (uint256 fundingFee) {
    fundingFee = PerpLib._calPercentage(
        (contractSize * price) / WEI_UNIT,
        _getFundingRate(isBuy, contractSize, price, underlyingToken)
    );
}
```

Listing H-04.5 The _getFundingRate and _getFundingFee functions of the PerpCoreBase contract

H-05 Inconsistency Between Calculated Global Unrealized PNL And Actual Value

Risk	Likelihood	Impact	Status
● High	● Medium	● High	Fixed
Locations	PerpTrading.sol::_updateGlobalStatus L: 650 - 664		

Detailed Issue

The `_updateGlobalStatus` function calculates `globalStats[underlying].averagePriceLong` and `globalStats[underlying].averagePriceShort.` These values are the average price of all positions, the calculation is shown in L631 - 636 in code snippet H-05.1.

However when a position is closed, there exists inconsistency between calculated global unrealized PNL and the actual value. Since after closing positions, the `averagePriceLong/Short` is not updated.

```
File: PerpTrading.sol
618:     function _updateGlobalStatus(
619:         address underlying,
620:         bool isOpen,
621:         bool isLong,
622:         uint256 contractSize,
623:         uint256 entryPrice,
624:         int256 pnl,
625:         int256 settlePnl
626:     ) internal {
627:         GlobalStat storage stat = globalStats[underlying];
628:
629:         // CASE: open position
630:         if (isOpen && isLong) {
631:             stat.averagePriceLong = PerpLib._calculateAverageValue(
632:                 stat.averagePriceLong,
633:                 stat.totalContractSizeLong,
634:                 entryPrice,
635:                 contractSize
636:             );
637:             stat.totalContractSizeLong += contractSize;
638:         }
639:         if (isOpen && !isLong) {
640:             stat.averagePriceShort = PerpLib._calculateAverageValue(
641:                 stat.averagePriceShort,
642:                 stat.totalContractSizeShort,
643:                 entryPrice,
644:                 contractSize
645:             );
646:             stat.totalContractSizeShort += contractSize;
```

```
        }

        // CASE: close position
        if (!isOpen && isLong) {
            stat.totalContractSizeLong -= MathUpgradeable.min(
                contractSize,
                stat.totalContractSizeLong
            );
            if (stat.totalContractSizeLong == 0) stat.averagePriceLong = 0;
        }

        if (!isOpen && !isLong) {
            stat.totalContractSizeShort -= MathUpgradeable.min(
                contractSize,
                stat.totalContractSizeShort
            );
            if (stat.totalContractSizeShort == 0) stat.averagePriceShort = 0;
        }
```

Listing H-05.1 The `_updateGlobalStatus` function of the PerpTrading contract

Impact

Inconsistency between all unrealized PNL value from `_getAllUnRealizedPNL` function and the actual unrealized global PNL. This further leads to

1. Incorrect `atpPrice` and `traderPnl,` this causes malfunctions in `_deposit` (as shown in code snippet H-05.4) and `_withdraw` functions (as shown in code snippet H-05.5).
2. Incorrect liquidatable threshold in the `_isLiquidableByTotalPnl` function as shown in (as shown in code snippet H-05.6).

```
File: PerpCoreBase.sol
    function _getAllUnRealizedPNL() internal view virtual returns (int256 result) {
        (uint256[] memory prices, , bool isStale) = _queryPythPrices(allowUnderlyingList);
        require(!isStale, "PT/unrealize-pnl-stale");

        for (uint8 i = 0; i < allowUnderlyingList.length; i++) {
            GlobalStat memory temp = globalStats[allowUnderlyingList[i]];

            int256 longPNL = PerpLib._calculatePNL(
                temp.totalContractSizeLong,
                prices[i],
                temp.averagePriceLong
            );
            int256 shortPNL = PerpLib._calculatePNL(
                temp.totalContractSizeShort,
                temp.averagePriceLong,
```

```
                    prices[i]
                );
                result += (longPNL + shortPNL);
            }
        }
```

Listing H-05.2 The `_getAllUnRealizedPNL` function of the PerpCoreBase contract

```
File: PerpLending.sol
    function _getInterestTokenPrice() internal view returns (uint256 price) {
        int256 traderPnl = _getAllUnRealizedPNL();

        price = uint256(
            PerpLib._calculateInterestTokenPrice(
                int256(liquidity),
                int256((atpTokenTotalSupply * WEI_UNIT) / COLLATERAL_UNIT),
                traderPnl,
                int256(WEI_UNIT)
            )
        );
    }
```

Listing H-05.3 The `_getInterestTokenPrice` function of the PerpLending contract

```
File: PerpLending.sol
    function _deposit(uint256 nftId, uint256 depositAmount) internal {
        require(depositAmount > 0, "PT/deposit-amount-is-zero");

        nftId = _getUsableToken(msg.sender, nftId);
        uint256 atpPrice = _getInterestTokenPrice();
        require(atpPrice != 0, "PL/price-is-zero");

        //mint p, atp in 1eColla unit
        uint256 mintedP = _mintPToken(msg.sender, nftId, depositAmount);
        uint256 mintedAtp = _mintAtpToken(
            msg.sender,
            nftId,
            ((depositAmount * WEI_UNIT) / atpPrice),
            atpPrice
        );
```

Listing H-05.4 The `_deposit` function of the PerpLending contract

```
File: PerpLending.sol
    function _withdraw(uint256 nftId, uint256 withdrawAmount) internal {
        PoolTokens storage tokenHolder = tokenHolders[nftId];

        uint256 atpPrice = _getInterestTokenPrice();
        require(atpPrice != 0, "PL/price-is-zero");

        if (_getAllUnRealizedPNL() < 0)
            withdrawAmount = MathUpgradeable.min(
```

```
                (liquidity * COLLATERAL_UNIT) / WEI_UNIT,
                withdrawAmount
            );

        (uint256 pBurnAmount, uint256 atpBurnAmount) = _burnToken(
            msg.sender,
            nftId,
            withdrawAmount,
            atpPrice,
            tokenHolder
        );

        // withdrawAmount is in 1e18
        withdrawAmount = _updateLiquidity((atpBurnAmount * atpPrice) / WEI_UNIT,
false);
```

Listing H-05.5 The `_withdraw` function of the PerpLending contract

```
File: PerpTrading.sol
    function _isLiquidableByTotalPnl() internal view returns (bool
isPosLiquidable) {
        int256 totalPnl = _getAllUnRealizedPNL();
        isPosLiquidable =
            (liquidity * liquidatePnlRatio) / WEI_PERCENT_UNIT <=
PerpLib._toUint(totalPnl);
    }
```

Listing H-05.6 The `_isLiquidableByTotalPnl` function of the PerpTrading contract

Scenario

To demonstrate the inconsistency between all unrealized PNL from `_getAllUnRealizedPNL` function and the actual unrealized global PNL.

Consider the following scenario:

1. There are no positions opening.
2. Alice opens a long WAVAX/USDC position with a 1e18 contract size, the current price is 1 WAVAX = 10 USD.

   `globalStats[underlying].averagePriceLong` is updated to ((1e18*10e18) + (0*0)) / (1e18+0)

   `globalStats[underlying].averagePriceLong` = 10e18.
3. Bob opens a long WAVAX/USDC position with a 1e18 contract size, the current price is 1 WAVAX = 90 USD.

   `globalStats[underlying].averagePriceLong` is updated to ((1e18*90e18) + (1e18*10e18)

) / (1e18+1e18)

`globalStats[underlying].averagePriceLong` = 50e18.

4. Now the current price is 1 WAVAX = 200 USD.
5. If all positions are closed, the unrealized trader PNL can be calculated from `_getAllUnRealizedPNL()` as shown in code snippet H-05.2. Note that, we focus on longPNL since there is another bug in shortPNL calculation.

   The calculated long PNL is

```
int256 longPNL = PerpLib._calculatePNL(
        temp.totalContractSizeLong,
        prices[i],
        temp.averagePriceLong
);
temp.totalContractSizeLong = 2e18
prices[i] = 200e18
temp.averagePriceLong = 50e18
```

   so, calculated unrealized longPNL is (2e18 * (200e18 - 50e18)) / 1e18 = 300e18.

6. **At this step, all unrealized longPNL value still matches the actual PNL if all positions are closed.**

   if Alice closes all her positions, she will get the profit as

```
PerpLib._calculatePNL(args.closingSize, args.currentPrice, pos.entryPrice)
args.closingSize = 1e18
args.currentPrice = 200e18
pos.entryPrice = 10e18
```

   Alice's profit will be (1e18 * (200e18 - 10e18)) / 1e18 = 190e18

   and if Bob closes all his positions, Bob's profit will be (1e18 * (200e18 - 90e18)) / 1e18 = 110e18

   total unrealized profit = 190e18 + 110e18 = 300e18 = longPNL of `_getAllUnRealizedPNL()`

   The total summary of all positions unrealized PNL still aligns with the longPNL value of `_getAllUnRealizedPNL()`.

7. Bob closes his position, `stat.totalContractSizeLong` is updated to 1e18 but `stat.averagePriceLong` is not updated. **Because `stat.totalContractSizeLong` != 0 as shown in L655 of code snippet H-05.1.**
8. Now the longPNL value of `_getAllUnRealizedPNL()` will be

```
int256 longPNL = PerpLib._calculatePNL(
        temp.totalContractSizeLong,
        prices[i],
        temp.averagePriceLong
);
temp.totalContractSizeLong = 1e18
```

`prices[i]` = 200e18

`temp.averagePriceLong` = 50e18

so, **calculated unrealized longPNL is (1e18 * (200e18 - 50e18)) / 1e18 = 150e18.**

9. **But this unrealized longPNL does not match with the actual PNL if all positions are closed.**

   Since if Alice closes all her positions, she will get the profit as

   `PerpLib._calculatePNL(args.closingSize, args.currentPrice, pos.entryPrice)`

   `args.closingSize` = 1e18

   `args.currentPrice` = 200e18

   `pos.entryPrice` = 10e18

   Alice profit will be (1e18 * (200e18 - 10e18)) / 1e18 = **190e18**

   As shown here, the longPNL value (150e18) is not close to the actual PNL if all positions are closed (190e18). This is because `stat.averagePriceLong` is not updated when Bob closes his position.

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend tracking lenders who lose their benefits and compensating them later.

Reassessment

The `FWX` team fixed this issue in commit: `543dbda`.

Medium Risk

M-01 Inconsistent Liquidation Check Across Function

Risk	Likelihood	Impact	Status
Medium	High	Low	Fixed
Locations	PerpTrading.sol::closePosition L: 56 PerpTrading.sol::_isLiquidable L: 484		

Detailed Issue

We found an inconsistent liquidation status check between the `closePosition` function and the `_isLiquidable` function as follows:

- `closePosition()`: isLiquidate = netBalance **<=** int256(nftStats[nftId].totalMaintenanceMargin);
- `_isLiquidable()`: isLiquidate = netBalance **<** int256(nftStat.totalMaintenanceMargin);

```
File: PerpTrading.sol
45:     function closePosition(
46:         uint256 nftId,
47:         uint256 posId,
48:         uint256 closingSize,
49:         bytes[] memory pythUpdateData
50:     ) external payable {
51:         nftId = _getUsableToken(msg.sender, nftId);
52:         _updatePythData(nftId, pythUpdateData);
53:         bool isLiquidate;
54:         {
55:             (int256 netBalance, ) = _getBalance(nftId);
56:             isLiquidate = netBalance <=
int256(nftStats[nftId].totalMaintenanceMargin);
57:         }
58:         _closePosition(nftId, posId, closingSize, isLiquidate);
59:     }
```

Listing M-01.1 The `closePosition` function of the `PerpTrading` contract

```
File: PerpTrading.sol
474:     function _isLiquidable(
475:         uint256 nftId,
476:         uint256 posId
477:     ) internal view returns (bool isPosLiquidable) {
478:         PositionState storage posState = positionStates[nftId][posId];
479:         Position storage pos = positions[nftId][posState.pairByte];
```

```
        require(pos.id != 0 && posState.active, "PT/position-already-closed");

        (int256 netBalance, ) = _getBalance(nftId);
        NftStat memory nftStat = nftStats[nftId];
        return netBalance < int256(nftStat.totalMaintenanceMargin);
    }
```

Listing M-01.2 The _isLiquidable function of the PerpTrading contract

Impact

This inconsistency can lead to scenarios where a position might be considered liquidatable in one function but not in another.

Recommendations

We recommend using the _isLiquidable function within the closePosition function to ensure consistent internal logic when validating whether a position is liquidatable.

```
File: PerpTrading.sol
    function closePosition(
        uint256 nftId,
        uint256 posId,
        uint256 closingSize,
        bytes[] memory pythUpdateData
    ) external payable {
        nftId = _getUsableToken(msg.sender, nftId);
        _updatePythData(nftId, pythUpdateData);
        bool isLiquidate;
        {
            isLiquidate = _isLiquidable(nftId, posId);
        }
        _closePosition(nftId, posId, closingSize, isLiquidate);
    }
```

Listing M-01.3 The improved closePosition function of the PerpTrading contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The FWX team fixed this issue by removing the isLiquidate check from the closePosition function.

M-02 Potential Use Of Stale Oracle Data In Deposit Function

Risk	Likelihood	Impact	Status
Medium	High	Low	**Fixed**
Locations	PerpLending.sol::deposit L: 8		

Detailed Issue

The `deposit` function relies on Pyth oracle data to calculate the mint amount of `atpToken`. To prevent using stale price data, its internal function includes a check to revert if the prices are outdated as shown in code snippet M-02.1.

However, the `deposit` function currently does not include `pythUpdateData` as input to update Pyth data before fetching prices.

```
File: PerpCoreBase.sol
100:     function _getAllUnRealizedPNL() internal view virtual returns (int256
result) {
101:         (uint256[] memory prices, , bool isStale) =
_queryPythPrices(allowUnderlyingList);
102:         require(!isStale, "PT/unrealize-pnl-stale");
```

Listing M-02.1 The internal `_getAllUnRealizedPNL` function used by the `deposit` function

Impact

This increases the risk of using stale price data, leading to failed deposit transactions.

Recommendations

We recommend that the `FWX` team add `pythUpdateData` as a parameter of the deposit function. This will reduce the risk of fetching stale price data.

```
File: PerpLending.sol
08:     function deposit(uint256 nftId, uint256 amount, bytes[] memory
pythUpdateData) external payable {
09:         _updatePythData(nftId, pythUpdateData);
10:         return _deposit(nftId, amount);
11:     }
```

Listing M-02.2 The improved `deposit` function of the `PerpLending` contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

M-03 Improper Handling Of Edge Cases in Pyth Price Calculation Function

Risk	Likelihood	Impact	Status
Medium	Low	High	**Fixed**
Locations	PerpLib.sol::_calculatePythPrice L: 100 - 109		

Detailed Issue

We found improper handling of edge cases in the `_calculatePythPrice` function in the `PerpLib` contract. This function silently returns 0 when it encounters invalid input (non-negative exponent or negative price) instead of reverting or signaling an error. This can lead to zero prices being propagated through the system, potentially causing issues in price-dependent calculations throughout the protocol.

```
File: PerpLib.sol
101:   function _calculatePythPrice(
102:       PythStructs.Price memory _price
103:   ) internal pure returns (uint256 price) {
104:       if (_price.expo >= 0 || _price.price < 0) return 0;
105:
106:       uint32 uExpo = uint32(-_price.expo);
107:       uint64 uPrice = uint64(_price.price);
108:       price = (uPrice * (10 ** (18 - uExpo)));
109:   }
```

Listing M-03.1 The `_calculatePythPrice` function of the PerpLib contract

Impact

The impact of this issue can lead to several problematic cases:

- Mispricing: Positions could be opened or closed at incorrect prices, leading to unfair gains or losses.
- Incorrect risk assessment: The protocol's risk calculations (e.g., liquidation thresholds, collateral requirements) could be inaccurate.
- Potential exploitation: Malicious actors could potentially take advantage of zero prices to open large positions with minimal collateral.
- System instability: Global stats and liquidity ratios could be incorrectly calculated, affecting the protocol's overall stability.
- Unfair liquidations: Positions might be unfairly liquidated or protected from liquidation due to incorrect price information.

- Silent failures: Zero prices could propagate through the system without triggering obvious errors, making issues harder to detect and debug.

These cases could result in financial losses for users and the protocol, as well as undermine the system's integrity and user trust.

Recommendations

We recommended that the function should revert when encountering invalid price data instead of returning 0, as we confirmed with the FWX team. This change ensures that the function will revert if the exponent is non-negative or if the price is zero or negative, providing clear error signaling instead of silently returning 0.

```
File: PerpLib.sol
 function _calculatePythPrice(
    PythStructs.Price memory _price
) internal pure returns (uint256 price) {
    require(_price.expo < 0 && _price.price > 0, "Invalid Pyth price data");

    uint32 uExpo = uint32(-_price.expo);
    uint64 uPrice = uint64(_price.price);
    price = (uPrice * (10 ** (18 - uExpo)));
}
```

Listing M-03.2 The improved `_calculatePythPrice` function of the PerpLib contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The FWX team adopted our recommendation and fixed this issue.

M-04	Donation Attack To Increase atpPrice

Risk	Likelihood	Impact	Status
● Medium	● Low	● High	**Acknowledged**
Locations	PerpLending.sol::_deposit L: 39		

Detailed Issue

In the `PerpLending` contract, lenders receive `atpTokens` proportional to their deposits. The contract calculates the `atpPrice`, which determines the ratio of the depositing token to the `atpTokenTotalSupply` and the unrealized `traderPnl`. This `atpPrice` is then used to mint `atpTokens` for the lenders.

However, we've identified a vulnerability where attackers can manipulate the `atpPrice`, inflating it to a level that prevents other lenders from receiving any `atpTokens`.

Impact

Attackers can prevent other lenders from getting **`atpToken`** from their deposits.

Exploit Scenario

1. The collateral token is WETHe (for this proof of concept, we use 18 decimal tokens as collateral to avoid a calculation bug in the withdraw function)
2. There are no deposits initially.
3. The attacker deposits 1,000e18 WETHe.
4. A trader performs a sequence of operations (`depositCollateral`, `openPosition`, and `closePosition`) resulting in some profit for the lenders and an increase in the `atpPrice`.
5. The attacker reduces the `atpTokenTotalSupply` to nearly 0.

   He can achieve this by calling the withdraw function with a specific `withdrawAmount`.

   This `withdrawAmount` is calculated so that the `atpBurnAmount` is just slightly less than the tokenHolder's `atpToken` balance

   This `withdrawAmount` must result in `atpBurnAmount` slightly less than tokenHolder.atpToken.

   Since `atpBurnAmount` can be calculated from (`withdrawAmount` * `WEI_UNIT`) / `atpPrice` as shown in code snippet M-04.1.

   The attacker can find the specific `withdrawAmount` as:

a. `atpBurnAmount` = `tokenHolder.atpToken` - 1
b. Thus, (`withdrawAmount` * `WEI_UNIT`) / `atpPrice` = `tokenHolder.atpToken` - 1
c. Therefore, `withdrawAmount` = (`tokenHolder.atpToken` - 1) * `atpPrice` / `WEI_UNIT`

Using this calculated `withdrawAmount`, the `atpTokenTotalSupply` is reduced to just 2 tokens. Despite the `atpTokenTotalSupply` being extremely low, some liquidity from lender profit remains in the smart contract, this results in high `atpPrice` with low `atpTokenTotalSupply.`

6. At this point, the attacker can inject more liquidity into the system without increasing the `atpTokenTotalSupply` by repeatedly calling the deposit function with a specific `depositAmount`.

   Since `mintedAtp` = (`depositAmount` * `WEI_UNIT`) / `atpPrice` as shown in code snippet M-04.2, and we want `mintedAtp` to be zero, we can find the specific `depositAmount` as follows:

   a. `depositAmount` * `WEI_UNIT` must be less than `atpPrice`
   b. we choose, `depositAmount` * `WEI_UNIT` = `atpPrice` - 1
   c. So, `depositAmount` = (`atpPrice` - 1) / `WEI_UNIT`

   Given that `atpPrice` is calculated as ((`liquidity` - `traderPnl`) * `WEI_UNIT`) / `atpTokenTotalSupply` and `traderPnl` = 0, we can further refine this to:

   `depositAmount` = ((`liquidity` * `WEI_UNIT`) / `atpTokenTotalSupply` - 1) / `WEI_UNIT`

   As a result, each deposit increases liquidity without altering the `atpTokenTotalSupply`. This means that by repeatedly calling the `deposit` function, the `atpPrice` can be inflated dramatically. For example, **after 40 iterations, the `atpPrice` can rise from 122e18 to 2,035,466,886e18.**

7. Now, the `atpPrice` is significantly high enough to cause several unexpected behaviors including

   a. Other lenders will not get any `atpToken` minted with their regular deposit amount. For example, if lender A deposits 1,000e18 WETHe to the pool, he will get 0 `atpToken` because `mintedAtp` = (1,000e18 * `WEI_UNIT`) / 2,035,466,886e18 = 0
   b. The attacker can benefit from other lenders. Other lender liquidity will be used by traders, but the attacker will be the only lender who gains lender profit. Because other lenders do not have any `atpToken` to claim their profit.

```
File: PerpLending.sol
    function _burnToken(
        address receiver,
        uint256 nftId,
        uint256 withdrawAmount,
        uint256 atpPrice,
        PoolTokens memory tokenHolder
    ) internal returns (uint256 pBurnAmount, uint256 atpBurnAmount) {
        atpBurnAmount = MathUpgradeable.min(
            tokenHolder.atpToken,
```

```
            (withdrawAmount * WEI_UNIT) / atpPrice
        );
        atpBurnAmount = _burnAtpToken(receiver, nftId, atpBurnAmount, atpPrice);
```

Listing M-04.1 The `_burnToken` function of the PerpLending contract

```
File: PerpLending.sol
    function _deposit(uint256 nftId, uint256 depositAmount) internal {
        require(depositAmount > 0, "PT/deposit-amount-is-zero");

        nftId = _getUsableToken(msg.sender, nftId);
        uint256 atpPrice = _getInterestTokenPrice();
        require(atpPrice != 0, "PL/price-is-zero");

        //mint p, atp in 1eColla unit
        uint256 mintedP = _mintPToken(msg.sender, nftId, depositAmount);
        uint256 mintedAtp = _mintAtpToken(
            msg.sender,
            nftId,
            ((depositAmount * WEI_UNIT) / atpPrice),
            atpPrice
        );
```

Listing M-04.2 The `_deposit` function of the PerpLending contract

Recommendations

We recommend the FWX team prevent this attack by keeping some collateral tokens in the pool.

This can be done by either

1. The FWX team must be the first depositor and deposit a fixed amount of tokens in the pool. Note that, this deposited token always needs to be in the pool to significantly increase the tokens the attacker needs to inflate the atpPrice.
2. The protocol could split a fixed amount of the first depositor's tokens into the deposit amount of address zero (similar to Uniswap's approach). This will ensure that a fixed amount of tokens is always in the pool. Therefore, it will significantly increase the tokens the attacker needs to inflate the atpPrice.

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team has acknowledged this issue. As a solution, `FWX` will initially provide liquidity with a significant amount locked in the pool permanently; this measure is designed to prevent potential inflation attacks.

M-05 Lack of Price Protection Mechanisms In Position Opening

Risk	Likelihood	Impact	Status
● Medium	● Low	● High	**Acknowledged**
Locations	PerpTrading.sol::openPosition L: 31		

Detailed Issue

The `openPosition` function (as shown in code snippet M-05.1) allows traders to open future positions. However, there are some crucial checks missing:

1. **Slippage:** Given the highly volatile nature of the crypto and DeFi space, the function does not confirm whether the trader is satisfied with the entry price of the position.
2. **Expiration Timestamp Check:** The `openPosition` function lacks a safeguard to ensure that the transaction has not been pending in the mempool for too long. As a result, a trader could unintentionally open an outdated position that no longer meets their expectations.

```
File: PerpTrading.sol
31:     function openPosition(
32:         uint256 nftId,
33:         bool isLong,
34:         address collateralToken,
35:         address underlyingToken,
36:         uint256 contractSize,
37:         uint256 leverage,
38:         bytes[] memory pythUpdateData
39:     ) external payable {
40:         nftId = _getUsableToken(msg.sender, nftId);
41:         _updatePythData(nftId, pythUpdateData);
42:         _openPosition(nftId, isLong, collateralToken, underlyingToken,
contractSize, leverage);
43:     }
```

Listing M-05.1 The `openPosition` function of the PerpTrading contract

Impact

Traders can not specify their acceptable slippage and expiration for their position.

Recommendations

We recommend that the `FWX` team add `slippage` and `deadline` as parameters to the `openPosition` function. Additionally, a maximum slippage limit could be enforced by the global configurations to safeguard traders against unexpected price volatility.

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team has acknowledged this issue. The statement is: "For this version, we will continue without slippage. We will add slippage when we implement the next release of `Perpetual Trading`. For the deadline, we have decided to use `stalePeriod` as deadline."

M-06 The totalMaintenanceMargin State Is Not Reset When All Positions Are Closed

Risk	Likelihood	Impact	Status
• Medium	• Medium	• Medium	Fixed
Locations	PerpTrading.sol::_closePosition L: 308 - 472		

Detailed Issue

The `nftStat.totalMaintenanceMargin` state (L484 in code snippet M-06.1) is the margin that determines whether a position is eligible for liquidation. However, we found that the `nftStat.totalMaintenanceMargin` state (L411 in code snippet M-06.2) may not be fully cleared when all positions are closed, due to rounding down from division (L339 in code snippet M-06.2). Consequently, the remaining value will accumulate when new positions are opened.

```
File: PerpTrading.sol
474:     function _isLiquidable(
475:         uint256 nftId,
476:         uint256 posId
477:     ) internal view returns (bool isPosLiquidable) {
478:         PositionState storage posState = positionStates[nftId][posId];
479:         Position storage pos = positions[nftId][posState.pairByte];
480:         require(pos.id != 0 && posState.active, "PT/position-already-closed");
481:
482:         (int256 netBalance, ) = _getBalance(nftId);
483:         NftStat memory nftStat = nftStats[nftId];
484:         return netBalance < int256(nftStat.totalMaintenanceMargin);
485:     }
```

Listing M-06.1 The `_isLiquidable` function of the PerpTrading contract

```
File: PerpTrading.sol
308:     function _closePosition(
309:         uint256 nftId,
310:         uint256 posId,
311:         uint256 closingSize,
312:         bool isLiquidate
313:     ) internal returns (ClosePositionArgs memory args) {
314:         PositionState storage posState = positionStates[nftId][posId];
315:         Position storage pos = positions[nftId][posState.pairByte];
316:         NftStat storage nftStat = nftStats[nftId];
317:         require(pos.id != 0 && posState.active, "PT/position-already-closed");
318:
319:         args.closingSize = MathUpgradeable.min(closingSize, pos.contractSize);
320:         {
321:             uint64 publishTime;
322:             (args.currentPrice, publishTime) =
```

```
_getSafeCurrentPrice(pos.underlyingToken);

            args.currentPrice = PerpLib._calculatePriceWithSpread(
                args.currentPrice,
                args.closingSize,
                !posState.isLong,
                publishTime,
                stalePeriod,
                spreadNotional[pos.underlyingToken],
                spread[pos.underlyingToken]
            );
        }

        // Calculate variable
        args.notional = (args.closingSize * args.currentPrice) / WEI_UNIT;
        args.collateral = (pos.collateralLocked * args.closingSize) /
pos.contractSize;
        args.maintenanceMargin = PerpLib._calPercentage(
            (args.closingSize * pos.entryPrice) / WEI_UNIT,
            maintainanceMarginRatio[pos.underlyingToken]
        );
---
        nftStat.totalCollateralLocked -= args.collateral;
        if (args.maintenanceMargin > nftStat.totalMaintenanceMargin) {
            nftStat.totalMaintenanceMargin = 0;
        } else {
            nftStat.totalMaintenanceMargin -= args.maintenanceMargin;
        }

        // Note: set to zero if all position are closed
        if (nftStat.totalCollateralLocked == 0) {
            nftStat.unsettlePnl = 0;
        }
---
        return args;
    }
```

Listing M-06.2 The `_closePosition` function of the PerpTrading contract

Impact

The position will be closer to the liquidation criteria than it should be.

Recommendations

We recommend resetting the `nftStat.totalMaintenanceMargin` when all positions are completely closed by using the `nftStat.totalCollateralLocked` for check as shown in the code snippet below.

```
File: PerpTrading.sol
    function _closePosition(
        uint256 nftId,
        uint256 posId,
        uint256 closingSize,
        bool isLiquidate
    ) internal returns (ClosePositionArgs memory args) {
---
        // Note: set to zero if all position are closed
        if (nftStat.totalCollateralLocked == 0) {
            nftStat.unsettlePnl = 0;
            nftStat.totalMaintenanceMargin = 0;
        }
---
        return args;
    }
```

Listing M-06.3 The improved `_closePosition` function of the PerpTrading contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

M-07 Improperly Handle Liquidation Case When Closing Position

Risk	Likelihood	Impact	Status
● Medium	● Low	● High	**Acknowledged**
Locations	PerpTrading.sol::closePosition L: 45 - 59		

Detailed Issue

Traders can close their positions at will. However, if a position meets the liquidation criteria, the protocol empowers liquidators to forcibly close it. This mechanism protects both the protocol and lenders from potential losses of profit.

In the position closing process, the `closePosition` function has the liquidation check (L56 in code snippet M-07.1) before invoking the `_closePosition` function (L58 in code snippet M-07.1).

However, we discovered that the `_closePosition` function merely emits an event without performing the liquidation process. This allows traders to preemptively close their positions before liquidators can perform the liquidation process.

```
File: PerpTrading.sol
45:     function closePosition(
46:         uint256 nftId,
47:         uint256 posId,
48:         uint256 closingSize,
49:         bytes[] memory pythUpdateData
50:     ) external payable {
51:         nftId = _getUsableToken(msg.sender, nftId);
52:         _updatePythData(nftId, pythUpdateData);
53:         bool isLiquidate;
54:         {
55:             (int256 netBalance, ) = _getBalance(nftId);
56:             isLiquidate = netBalance <=
int256(nftStats[nftId].totalMaintenanceMargin);
57:         }
58:         _closePosition(nftId, posId, closingSize, isLiquidate);
59:     }
```

Listing M-07.1 The `closePosition` function of the PerpTrading contract

```
File: PerpTrading.sol
308:     function _closePosition(
309:         uint256 nftId,
310:         uint256 posId,
311:         uint256 closingSize,
312:         bool isLiquidate
```

```
    ) internal returns (ClosePositionArgs memory args) {
        PositionState storage posState = positionStates[nftId][posId];
        Position storage pos = positions[nftId][posState.pairByte];
        NftStat storage nftStat = nftStats[nftId];
        require(pos.id != 0 && posState.active, "PT/position-already-closed");
---
        if (isLiquidate) {
            emit LiquidatePosition(
                _getTokenOwnership(nftId),
                nftId,
                posId,
                posState.isLong,
                msg.sender,
                args.closingSize,
                args.currentPrice,
                posState.pairByte,
                address(0)
            );
        }
---
        return args;
    }
```

Listing M-07.2 The `_closePosition` function of the PerpTrading contract

Impact

Traders may close their positions even when it should be liquidated, potentially disadvantaging the protocol and lenders by depriving them of fees they should have received.

Recommendations

For the recommendation, we advise the `FWX` team to handle the liquidate status properly when closing positions. This will prevent the protocol and lenders from being deprived of fees they should have received.

Reassessment

The `FWX` team has acknowledged this issue. The statement is: "`FWX` enables users to close their positions even when they are at risk of liquidation. Users have the option to call either the `liquidate` or `closePosition` functions. However, `FWX` recommends using the `closePosition` function to avoid incurring protocol fees that would be charged if the liquidate function is called. While anyone can call the `liquidate` function due to the protocol's design, users are encouraged to close their positions manually to minimize costs."

M-08 Improper Price Usage for Calculating Trading Fees

Risk	Likelihood	Impact	Status
Medium	Medium	Medium	Acknowledged
Locations	PerpTrading.sol::_openPosition L: 223 - 226 PerpTrading.sol::_closePosition L: 342 - 345 PerpTrading.sol::_getUnrealizedPnlAndFee L: 811 - 813		

Detailed Issue

The protocol currently uses the perturbed notional (`size * perturbed price`) when calculating the trading fee. This approach can lead to inaccuracies, particularly when large positions are closed, causing significant price perturbations.

Typically, trading fees are calculated based on the current spot price. This is because trading fees are often meant to be a straightforward reflection of the transaction value, and using the spot price keeps the fee structure simple and transparent.

The perturbed price reflects a temporary and self-induced market condition rather than the true market value. As a result, this can lead to consistent and potentially fair fee calculations.

```
File: PerpTrading.sol
155:     function _openPosition(
156:         uint256 nftId,
157:         bool isLong,
158:         address collateralToken,
159:         address underlyingToken,
160:         uint256 contractSize,
161:         uint256 leverage
162:     ) internal {
163:         PerpLib.OpenPositionTempStruct memory temp;
164:         {
165:             uint64 publishTime;
166:             (temp.entryPrice, publishTime) =
_getSafeCurrentPrice(underlyingToken);
167:             temp.entryPrice = PerpLib._calculatePriceWithSpread(
168:                 temp.entryPrice,
169:                 contractSize,
170:                 isLong,
171:                 publishTime,
172:                 stalePeriod,
173:                 spreadNotional[underlyingToken],
174:                 spread[underlyingToken]
175:             );
176:         }
---
```

```
        // ! open position (new and more)
        temp.contractSize = contractSize;
        temp.isLong = isLong;
        temp.leverage = leverage;
        temp.underlyingToken = underlyingToken;
        temp.notional = (temp.contractSize * temp.entryPrice) / WEI_UNIT;
        temp.collateral = (temp.notional * WEI_UNIT) / leverage;
        temp.maintenanceMargin = PerpLib._calPercentage(
            temp.notional,
            maintainanceMarginRatio[temp.underlyingToken]
        );

        temp.tradingFee = PerpLib._calPercentage(
            temp.notional,
            tradingFeeRates[temp.underlyingToken]
        );
```

Listing M-08.1 The _openPosition function of the PerpTrading contract

```
File: PerpTrading.sol
    function _closePosition(
        uint256 nftId,
        uint256 posId,
        uint256 closingSize,
        bool isLiquidate
    ) internal returns (ClosePositionArgs memory args) {
        PositionState storage posState = positionStates[nftId][posId];
        Position storage pos = positions[nftId][posState.pairByte];
        NftStat storage nftStat = nftStats[nftId];
        require(pos.id != 0 && posState.active, "PT/position-already-closed");

        args.closingSize = MathUpgradeable.min(closingSize, pos.contractSize);
        {
            uint64 publishTime;
            (args.currentPrice, publishTime) = _getSafeCurrentPrice(pos.underlyingToken);

            args.currentPrice = PerpLib._calculatePriceWithSpread(
                args.currentPrice,
                args.closingSize,
                !posState.isLong,
                publishTime,
                stalePeriod,
                spreadNotional[pos.underlyingToken],
                spread[pos.underlyingToken]
            );
        }

---
        // Calculate variable
        args.notional = (args.closingSize * args.currentPrice) / WEI_UNIT;
```

```
        args.collateral = (pos.collateralLocked * args.closingSize) / pos.contractSize;
        args.maintenanceMargin = PerpLib._calPercentage(
            (args.closingSize * pos.entryPrice) / WEI_UNIT,
            maintainanceMarginRatio[pos.underlyingToken]
        );
        args.tradingFee = PerpLib._calPercentage(
            args.notional,
            tradingFeeRates[pos.underlyingToken]
        );
```

Listing M-08.2 The `_closePosition` function of the `PerpTrading` contract

```
File: PerpTrading.sol
    function _getUnrealizedPnlAndFee(
        uint256 nftId
    ) internal view returns (int256 unrealizedPnl, uint256 tradingFee, uint256 fundingFee) {
---
            uint256 perturbPrice = PerpLib._calculatePriceWithSpread(
                prices[i],
                pos.contractSize,
                !posState.isLong,
                publishTimes[i],
                stalePeriod,
                spreadNotional[pos.underlyingToken],
                spread[pos.underlyingToken]
            );
---
            tradingFee +=
                (perturbPrice * pos.contractSize * tradingFeeRates[pos.underlyingToken]) /
                (WEI_UNIT * WEI_PERCENT_UNIT);
```

Listing M-08.3 The `_getUnrealizedPnlAndFee` function of the `PerpTrading` contract

Impact

Using the perturbed price for trading fee calculations can result in either **undercharging** or **overcharging** fees, depending on the direction of the price movement.

This inconsistency undermines the fairness of the platform, as users may not be charged fees that accurately reflect the market conditions.

Scenario

Consider a scenario where a user closes a large long position:

- Current Market Price: $100
- Perturbed Price Closing: $95
- Trading Fee Rate: 0.1%

If the trading fee is calculated based on the **perturbed price**:

Trading Fee: 95 × 0.1% = 0.095 USD

If the trading fee is calculated based on the **current market price**:

Trading Fee: 100 × 0.1% = 0.10 USD

This discrepancy results in the protocol collecting less fee revenue than it should, and the user benefits from an inaccurate fee calculation.

Recommendations

We recommend using the current market price rather than the perturbed price when calculating the notional for trading fees to ensure accuracy and fairness.

Additionally, this consideration should extend to other processes such as funding fee calculations and liquidity ratio validations, where it is crucial to evaluate which pricing approach better reflects the true cost and impact of large trades on market conditions.

Reassessment

The `FWX` team has acknowledged this issue. The statement is: "Worked as design. Our design for perturbed price is used as spread. So this is the final price used."

M-09 Over-Perturbed Entry Price Due To Initial Contract Size

Risk	Likelihood	Impact	Status
● Medium	● Medium	● Medium	Fixed
Locations	PerpTrading.sol::_openPosition L: 153 - 277		

Detailed Issue

The open trading position allows the opening of an opposite side position by implementing functionality to partially or fully close the current position (L188-L200). **When the opposite side position is opened with a size equal to or greater than the existing position, the new position will be opened with a size beyond the existing one.**

We discovered that **the entry price for opening an opposite side position can be affected by the initial contract size specified at L167.** Since the actual contract size is recalculated at L192, **the entry price is overly perturbed based on the initial size before this recalculation.**

This perturbed entry price will impact several factors, as it is used in various subsequent steps, including validating the opening input, calculating the notional and collateral, and determining fees.

```
File: PerpTrading.sol
155:     function _openPosition(
156:         uint256 nftId,
157:         bool isLong,
158:         address collateralToken,
159:         address underlyingToken,
160:         uint256 contractSize,
161:         uint256 leverage
162:     ) internal {
163:         PerpLib.OpenPositionTempStruct memory temp;
164:         {
165:             uint64 publishTime;
166:             (temp.entryPrice, publishTime) =
_getSafeCurrentPrice(underlyingToken);
167:             temp.entryPrice = PerpLib._calculatePriceWithSpread(
168:                 temp.entryPrice,
169:                 contractSize,
170:                 isLong,
171:                 publishTime,
172:                 stalePeriod,
173:                 spreadNotional[underlyingToken],
174:                 spread[underlyingToken]
175:             );
176:         }
---
182:         if (pos.id == 0) {
```

```
            temp.id = uint64(++currentPositionIndex[nftId]);
        } else {
            // ! position already exist
            temp.id = uint64(currentPositionIndex[nftId]);
            posState = positionStates[nftId][temp.id];
            if (posState.isLong != isLong) {
                _closePosition(nftId, pos.id, contractSize, false);
                if (contractSize > pos.contractSize) {
                    // ! close all open new
                    contractSize = contractSize - pos.contractSize;

                    temp.id = uint64(++currentPositionIndex[nftId]);
                    pos = positions[nftId][temp.pairByte];
                    posState = positionStates[nftId][temp.id];
                } else {
                    // ! close partial or close all and end
                    return;
                }
            }
        }
---
        _validateOpenPositionInput(
            collateralToken,
            underlyingToken,
            ((pos.contractSize * pos.entryPrice) + (contractSize * temp.entryPrice)) / WEI_UNIT,
            leverage
        );

        // ! open position (new and more)
        temp.contractSize = contractSize;
        temp.isLong = isLong;
        temp.leverage = leverage;
        temp.underlyingToken = underlyingToken;
        temp.notional = (temp.contractSize * temp.entryPrice) / WEI_UNIT;
        temp.collateral = (temp.notional * WEI_UNIT) / leverage;
        temp.maintenanceMargin = PerpLib._calPercentage(
            temp.notional,
            maintainanceMarginRatio[temp.underlyingToken]
        );
---
        _updateOpenPosition(nftId, temp);
---
    }
```

Listing M-09.1 The _openPosition function of the PerpTrading contract

Impact

The perturbed entry price leads to unfair trading conditions. It affects the validation of inputs, notional and collateral calculations, and fee determinations, potentially resulting in unexpected costs and financial risks for traders.

Scenario

Suppose that opening an opposite side position can happen in two ways:

1. **Open opposite position with a size greater than the existing position:**
    a. Existing Long position size 100e18
    b. Open new Short size 110e18

    For this case, the price in each step will be:

    c. Close ALL Long Position
    (perturbed existPrice with size: 100e18)
    d. Open new Short position with size 10e18 (110e18 - 100e18)
    (perturbed entryPrice with size: 110e18)

2. **Close the entire existing position and open the opposite side in separate transactions:**
    a. Existing Long position size 100e18
    b. Close All existing Long position size 100e18
    c. Open new Short size 10e18

    For this case, the implicit combination of (step 2.b) and (step 2.c) can be equivalent to step (1.b), and the price in each step will be:

    d. Close ALL Long Position (step 2.b)
    (perturbed existPrice with size: 100e18)
    e. Open new Short position with size 10e18
    (perturbed entryPrice with size: 10e18)

The scenario compares two methods for opening an opposite side position: one involves opening a larger short position while closing an existing long position, causing a significant perturbation based on the full size; the other involves closing the long position first and then opening a smaller short position, with different perturbation effects. Both methods can yield similar results but impact entry prices differently.

Recommendations

We recommend calculating the perturbed entry price based on the actual contract size to be opened by moving the entry price calculation to occur after the contract size has been recalculated for the opening opposite side position.

```
File: PerpTrading.sol
    function _openPosition(
        uint256 nftId,
        bool isLong,
        address collateralToken,
        address underlyingToken,
        uint256 contractSize,
        uint256 leverage
    ) internal {
        PerpLib.OpenPositionTempStruct memory temp;
---
        if (pos.id == 0) {
            temp.id = uint64(++currentPositionIndex[nftId]);
        } else {
            // ! position already exist
            temp.id = uint64(currentPositionIndex[nftId]);
            posState = positionStates[nftId][temp.id];
            if (posState.isLong != isLong) {
                _closePosition(nftId, pos.id, contractSize, false);
                if (contractSize > pos.contractSize) {
                    // ! close all open new
                    contractSize = contractSize - pos.contractSize;

                    temp.id = uint64(++currentPositionIndex[nftId]);
                    pos = positions[nftId][temp.pairByte];
                    posState = positionStates[nftId][temp.id];
                } else {
                    // ! close partial or close all and end
                    return;
                }
            }
        }
---

        {
            uint64 publishTime;
            (temp.entryPrice, publishTime) = _getSafeCurrentPrice(underlyingToken);
            temp.entryPrice = PerpLib._calculatePriceWithSpread(
                temp.entryPrice,
                contractSize,   // represents the contract size after recalculated (in case of opening opposite position)
                isLong,
                publishTime,
                stalePeriod,
                spreadNotional[underlyingToken],
                spread[underlyingToken]
```

```
            );
        }
```

Listing M-09.2 The improved `_openPosition` function of the PerpTrading contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

Low Risk

L-01 Recommend Adhering To Best Practices For Confidence Interval Validation In Pyth Network Integration

Risk	Likelihood	Impact	Status
• Low	• Low	• Medium	**Acknowledged**
Locations	PerpCoreBase.sol::_queryPythPrice L: 196 - 205		

Detailed Issue

The current implementation of the `_queryPythPrice` function in the protocol does not validate the confidence intervals of price data retrieved from the Pyth Network (`price.conf`).

The Pyth Network documentation (https://docs.pyth.network/price-feeds/best-practices) emphasizes validating confidence intervals to ensure data reliability. In scenarios where confidence intervals are disjoint or significantly divergent, failing to validate these intervals can lead to accepting price data with high uncertainty. This validation is crucial to prevent issues arising from highly uncertain price feeds.

Such cases, although rare, can lead to unreliable price data and fail to adhere to best practices for price feed validation.

```
File: PerpCoreBase.sol
196:     function _queryPythPrice(
197:         address token
198:     ) internal view virtual returns (uint256 price, uint64 publishTime, bool
isStale) {
199:         PythStructs.Price memory pythPrice =
pyth.getPriceUnsafe(pythOracleId[token]);
200:         price = PerpLib._calculatePythPrice(pythPrice);
201:         publishTime = uint64(pythPrice.publishTime);
202:         if (block.timestamp > stalePeriod + publishTime) {
203:             isStale = true;
204:         }
205:     }
```

Listing L-01.1 The `_queryPythPrice` function of the `PerpCoreBase` contract

```
File: node_modules/@pythnetwork/pyth-sdk-solidity/PythStructs.sol
contract PythStructs {
    // A price with a degree of uncertainty, represented as a price +- a
confidence interval.
    //
    // The confidence interval roughly corresponds to the standard error of a
```

```
normal distribution.
    // Both the price and confidence are stored in a fixed-point numeric
representation,
    // `x * (10^expo)`, where `expo` is the exponent.
    //
    // Please refer to the documentation at
https://docs.pyth.network/documentation/pythnet-price-feeds/best-practices for how
    // to how this price safely.
    struct Price {
        // Price
        int64 price;
        // Confidence interval around the price
        uint64 conf;
        // Price exponent
        int32 expo;
        // Unix timestamp describing when the price was published
        uint publishTime;
    }
```

Listing L-01.2 The `Price` struct of the `PythStructs` contract

Impact

The risk of relying on inaccurate price data. This can lead to financial losses from incorrect trades or liquidations, undermine user trust, and destabilize the protocol, affecting its overall stability and integrity.

Recommendations

We recommend adhering to best practices for handling price data by validating confidence intervals to ensure data reliability.

One effective approach is to use the σ/μ ratio, where μ represents the aggregate price and σ represents the aggregate confidence interval. This ratio helps measure the uncertainty of the price data: **the wider the confidence interval relative to the price, the higher the uncertainty**.

For example, if the aggregate price (μ) is \$100 and the confidence interval width is \$20, implying σ = \$10,

$$\sigma/\mu = 10/100 = 0.10 \text{ or } 10\%.$$

If the acceptable threshold for this ratio is 5%, a ratio of 10% indicates excessive uncertainty.

Reassessment

The `FWX` team has acknowledged this issue. The statement is: "For this version, we will continue without conf. We will apply `conf` when we implement the next release of `Perpetual Trading`."

L-02 Using Weak Source Of Randomness For Calculate Price With Spread

Risk	Likelihood	Impact	Status
● Low	● Low	● Medium	**Acknowledged**
Locations	PerpLib.sol::_getRandomNumber L: 210 - 218		

Detailed Issue

The `_calculatePriceWithSpread` function (code snippet L-02.1) used to calculate new price by using the randomness (L81 in code snippet L-02.1) within range of the `randomConfig` setting (L63 in code snippet L-02.1).

However, we noticed that the `_getRandomNumber` function (code snippet L-02.2) uses transaction and block information as a source of randomness (L213 in code snippet L-02.2). **This allows users to potentially predict the value by precomputing the randomness to obtain the minimum within the random range**.

```
File: PerpLib.sol
52:     function _calculatePriceWithSpread(
53:         uint256 price,
54:         uint256 size,
55:         bool isLong,
56:         uint64 publishTime,
57:         uint64 stalePeriod,
58:         uint256[] memory spreadNotional,
59:         uint256[] memory spread
60:     ) internal view returns (uint256 newPrice) {
61:         size = (price * size) / WEI_UNIT;
62:
63:         uint256[] memory randomConfig = new uint256[](2);
---
80:         timeMultiplier = MathUpgradeable.max(WEI_PERCENT_UNIT, timeMultiplier);
81:         uint256 randomness = _getRandomNumber(randomConfig[0], randomConfig[1]);
82:         randomness = (timeMultiplier * randomness) / WEI_PERCENT_UNIT;
83:
84:         if (isLong) {
85:             newPrice = (price * (WEI_PERCENT_UNIT + randomness)) /
WEI_PERCENT_UNIT;
86:         } else {
87:             newPrice = (price * (WEI_PERCENT_UNIT - randomness)) /
WEI_PERCENT_UNIT;
88:         }
89:     }
```

Listing L-02.1 The `_calculatePriceWithSpread` function of the PerpLib contract

```
File: PerpLib.sol
    function _getRandomNumber(uint256 min, uint256 max) internal view returns (uint256) {
        uint256 randomness = uint256(
            keccak256(
                abi.encodePacked(block.timestamp, gasleft(), tx.gasprice, msg.data, block.number)
            )
        );
        if (max - min == 0) return 0;
        return (randomness % (max - min)) + min;
    }
```

Listing L-02.2 The `_getRandomNumber` function of the PerpLib contract

Impact

This allows users to potentially predict the value by precomputing the randomness to obtain the minimum, enabling them to obtain the most advantageous price. **However, the impact of this issue is classified as "Medium" because the randomness result remains constrained within the admin-set range (L63 in code snippet L-02.1).**

Recommendations

We recommend the `FWX` team consider external sources of randomness via oracles like `Pyth Entropy` or `Chainlink VRF` instead.

Reassessment

The `FWX` team has acknowledged this issue. The statement is: "For this version, we will continue on existing random mechanisms. We will update to Pyth entropy when we implement the next release of `Perpetual Trading`"

L-03 Recommended Following Best Practices For Upgradeable Smart Contracts

Risk	Likelihood	Impact	Status
• Low	• Low	• Medium	**Fixed**
Locations	PerpCore.sol::constructor L: 11		

Detailed Issue

The `PerpCore` contract should enhance the disable initializer mechanism to be broadly supported in future upgrades and follow the best practices.

The practice above performs equivalent to `reinitializer(1)` which does not protect in the case of the contract upgrades that require reinitialization of the next version (version > 1).

```
File: PerpCore.sol
10: contract PerpCore is PerpLending, PerpSetting {
11:     constructor() initializer {}
```

Listing L-03.1 The disable initializer mechanism which does not protect in the case of the contract upgrades

Impact

This current implementation fails to safeguard against potential vulnerabilities during future upgrades that may require reinitialization with a higher version (version > 1).

As a result, the contract could be exposed to unintended reinitialization or misconfiguration in the next version of the contract, leading to potential security risks and system instability.

Recommendations

We recommend revising to use the `_disableInitializers` function.

The `_disableInitializers` function guards against future reinitializations by setting `_initialized` version to the max supported version (`uint8.max` for `OpenZeppelin` contract version `<= v4.9.5`, `uint64.max`, `>= v5.0.0`, for `OpenZeppelin` contract version).

```
File: PerpCore.sol
10: contract PerpCore is PerpLending, PerpSetting {
11:     constructor() {
12:         _disableInitializers();
```

```
13:     }
```

Listing L-03.2 The revising to use the `_disableInitializers` function

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

L-04 Incorrect Available Balance Validation For Open Position

Risk	Likelihood	Impact	Status
Low	Low	Medium	Fixed
Locations	PerpTrading.sol::_closePosition L: 236 - 240		

Detailed Issue

The protocol allows open positions only if the available balance is sufficient for required collateral and fees.

However, we found that the `_closePosition` function of the `PerpTrading` contract has an incorrect condition for validating the available balance in the opening process (L238 in the code snippet below), **This results in traders being unable to open positions when their available balance is exactly sufficient for collateral and fees.**

```
File: PerpTrading.sol
155:     function _openPosition(
156:         uint256 nftId,
157:         bool isLong,
158:         address collateralToken,
159:         address underlyingToken,
160:         uint256 contractSize,
161:         uint256 leverage
162:     ) internal {
---
236:         (, uint256 availableBalance) = _getBalance(nftId);
---
237:         require(
238:             availableBalance > temp.collateral + temp.tradingFee +
temp.fundingFee,
239:             "PT/insufficient-wallet-balance"
240:         );
---
278:     }
```

Listing L-04.1 The `_openPosition` function of the PerpTrading contract

Recommendations

We recommend re-implementing the mentioned validation to account for cases where the available balance is exactly sufficient for collateral and fees as shown in the code snippet below.

```
File: PerpTrading.sol
155:     function _openPosition(
```

```
        uint256 nftId,
        bool isLong,
        address collateralToken,
        address underlyingToken,
        uint256 contractSize,
        uint256 leverage
    ) internal {
---
        (, uint256 availableBalance) = _getBalance(nftId);
---
        require(
            availableBalance >= temp.collateral + temp.tradingFee + temp.fundingFee,
            "PT/insufficient-wallet-balance"
        );
---
    }
```

Listing L-04.2 The improved _openPosition function of the PerpTrading contract

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The FWX team adopted our recommendation and fixed this issue.

Informational Risk

I-01 Typo: "maintainance" Used Instead Of "maintenance" Throughout The Codebase

Risk	Likelihood	Impact	Status
Informational	Low	Low	**Fixed**
Locations	IPerpCore.sol IPerpCoreBase.sol PerpCoreBase.sol PerpSetting.sol PerpTrading.sol		

Detailed Issue

We found a consistent typo in the smart contracts where `"maintainance"` is used instead of the correct spelling `"maintenance"`. This typo appears in various state variables, functions, and events across the contracts.

We raise this issue because the term `"maintenance"` is particularly important in perpetual DeFi protocols, where it often refers to critical concepts like maintenance margin requirements. Using the correct terminology is crucial for clarity, consistency, and alignment with industry standards.

Affected instances:

1. State variables:
    - mapping(address => uint256) public maintainanceMarginRatio;
2. Functions:
    - setMarginRatio(address underlying, uint256 maintainance, uint256 minimum)
3. Events:
    - SetMaintainanceMarginRatio(msg.sender, underlying, oldValue, maintainance);
4. Other occurrences:
    - In various calculations and comparisons throughout the code

Impact

While this typo does not affect the functionality of the smart contracts, it impacts code readability. Correcting this typo will improve the overall quality and maintainability of the codebase.

Recommendations

We recommended replacing all instances of `"maintainance"` with `"maintenance"` to ensure correct spelling and improve code readability. This includes renaming variables, updating function parameters, and modifying event names and parameters.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

I-02 Unnecessary Dummy Price Functions And State Variable

Risk	Likelihood	Impact	Status
Informational	Low	Low	Acknowledged
Locations	IPerpCore.sol::setDummyPrice L: 228 IPerpCore.sol::dummyPrice L: 230 PerpCoreBase.sol::dummyPrice L: 92		

Detailed Issue

We found unnecessary dummy price functions in the `IPerpCore` interface and a corresponding state variable in the `PerpCoreBase` contract. These elements appear to be leftover development or testing code that should not be present in production-ready contracts. Their presence can confuse developers about the intended use of the contract, create unnecessary maintenance overhead, and potentially lead to misuse if mistakenly implemented or used in production code.

```
File: IPerpCore.sol
227:    // dummy
228:    function setDummyPrice(address token, uint256 newValue) external;
229:
230:    function dummyPrice(address token) external view returns (uint256);

File: PerpCoreBase.sol
92:    mapping(address => uint256) public dummyPrice;
```

Listing I-02.1 The `dummy` functions and state.

Recommendations

We recommend removing these functions from the `IPerpCore` interface and the corresponding state variable from the `PerpCoreBase` contract.

Reassessment

The `FWX` team has acknowledged this issue. The statement is: "Dummy is currently used at testing and we cannot find a solution for testing without a dummy price yet."

I-03 Recommended Removing Unused Code

Risk	Likelihood	Impact	Status
Informational	Low	Low	Fixed
Locations	PerpCore.sol L: 4 PerpCoreBase L: 7		

Detailed Issue

We found that unused codes can be removed for readability and maintainability as listed below.

- The PerpCore contract line 4
- The PerpCoreBase contract line 7

Recommendations

We recommend removing the unused codes to improve readability and maintainability of the protocol.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

I-04 Inconsistency Between Comment And Code

Risk	Likelihood	Impact	Status
Informational	Low	Low	Fixed
Locations	PerpLib.sol::_getFundingRate L: 119		

Detailed Issue

The comment in the `_getFundingRate` function states that `oi` will be negative when `totalOI` of short positions is lesser than long positions (as shown in code snippet I-05.1).

However, this comment is incorrect since `oi` will be negative when `totalOI` of short positions is greater than long positions.

```
File: PerpLib.sol
116:     function _getFundingRate(
117:         uint256[] memory fundingNetOI, // open interest of short or long
118:         uint256[] memory fundingRates,
119:         int256 oi, // + long > short, - short < long in % unit  (currentOIRatio)
120:         bool isBuy // isBuy -> open long, close short, !isBuy -> open short,
close long
121:     ) internal pure returns (uint256 fundingRate) {
```

Listing I-04.1 The `_getFundingRate` function of the PerpLib contract

Impact

The comment could mislead the code context.

Recommendations

We recommend that the `FWX` team correct the comment.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

I-05 Mismatch Between Interface And Implementation For setMinimumOpenSize Function

Risk	Likelihood	Impact	Status
Informational	Low	Low	Fixed
Locations	IPerpCore.sol::setMinimumOpenSize L: 209 PerpSetting.sol::setMinimumOpenSize L: 60 - 67		

Detailed Issue

We found that there is a discrepancy between the `setMinimumOpenSize` function declaration in the `IPerpCore` interface and its implementation in the `PerpSetting` contract. The interface specifies a return value of uint256, while the actual implementation does not return any value.

```
File: IPerpCore.sol
008: interface IPerpCore is IPerpCoreBase {
---
201:     function setMaximumOpenSize(address underlyingToken, uint256 value)
external;
---
209:     function setMinimumOpenSize(address underlyingToken, uint256 value) external
returns (uint256);
---
241: }
```

Listing I-05.1 The `setMinimumOpenSize` function in the `IPerpCore` interface.

```
File: PerpSetting.sol
007: contract PerpSetting is PerpCoreBase {
---
060:     function setMinimumOpenSize(
061:         address underlying,
062:         uint256 newValue
063:     ) external onlyConfigTimelockManager {
064:         uint256 oldValue = minimumOpenSize[underlying];
065:         minimumOpenSize[underlying] = newValue;
066:         emit SetMinimumOpenSize(msg.sender, underlying, oldValue, newValue);
067:     }
---
150: }
```

Listing I-05.2 The `setMinimumOpenSize` function in the `PerpSetting` contract.

Impact

This inconsistency can lead to compilation errors or unexpected behavior when interacting with the contract through its interface.

Recommendations

We recommend aligning the interface declaration with the actual implementation and maintaining consistency with similar functions like `setMaximumOpenSize`. Remove the returns (uint256) part from the `setMinimumOpenSize` function declaration in the `IPerpCore` interface:

```
File: IPerpCore.sol
008: interface IPerpCore is IPerpCoreBase {
---
201:     function setMaximumOpenSize(address underlyingToken, uint256 value)
external;
---
209:     function setMinimumOpenSize(address underlyingToken, uint256 value)
external;
---
241: }
```

Listing I-05.3 The improved `setMinimumOpenSize` function in the `IPerpCore` interface.

Important Note: The team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The `FWX` team adopted our recommendation and fixed this issue.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information

Contact	Link
Email	info@valix.io
Facebook	https://www.facebook.com/ValixConsulting
Twitter	https://twitter.com/ValixConsulting
Medium	https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/